



2008

**Proceedings of the
6th International Modelica Conference**

March 3rd – 4th, 2008
University of Applied Sciences Bielefeld,
Bielefeld, Germany

Bernhard Bachmann (editor)

Volume 1

organized by
The Modelica Association and
University of Applied Sciences Bielefeld

All papers of this conference can be downloaded from
<http://www.Modelica.org/events/modelica2008/>

Proceedings of Modelica' 2008

University of Applied Sciences Bielefeld,
Bielefeld, Germany, March 2008

Editor:

Prof. Dr. B. Bachmann

Published by:

The Modelica Association (<http://www.Modelica.org>) and
University of Applied Sciences Bielefeld (<http://www.fh-bielefeld.de>)

Printed by:

Printmedien Elbracht

Preface

The first International Modelica Conference took place in October 2000 in Lund, Sweden. Since then, Modelica has increasingly become the preferred language tool for physical modelling of complex systems. This is indicated by the high number of registrations from industry and science at the 6th International Modelica Conference held between 3rd and 4th March 2008 at the University of Applied Sciences, Bielefeld, Germany. It is also indicated by the number of excellent papers submitted to the program committee which made the task of selecting papers for oral and poster presentation very difficult and, last but not least, by the exhibition during the conference at which several companies will be represented. This volume contains the papers of the 68 oral presentations and 14 poster presentations at the conference. The ability of Modelica as a multi domain simulation language is demonstrated impressively by the various fields the papers are covering.

Due to the special features of the Modelica language, such as object-oriented modelling and the ability to reuse and exchange models, Modelica strongly supports an integrated engineering design process. Thus in various fields Modelica has become the standard tool for model exchange between suppliers and OEM's. A key issue for the success of Modelica is the continuous development of the Modelica language as well as the Modelica Standard Library under strict observance of compatibility to previous versions by the Modelica Association. The broad base of private and institutional members of the Modelica Association as a non-profit organization ensures language stability and security in software investments.

The 6th International Modelica conference was organized by the Modelica Association and by the University of Applied Sciences, Bielefeld, Germany. I would like to thank the local organizing committee, the technical program committee and the reviewers for offering their time and expertise throughout the organization of the conference. Together with the entire team of the local organizing committee I would like to wish all participants an excellent and fruitful conference.

Bielefeld, March 1st, 2008

Bernhard Bachmann

Program Chair

- Prof. Bernhard Bachmann, University of Applied Sciences Bielefeld, Bielefeld, Germany

Program Board

- Prof. Martin Otter, DLR, Oberpfaffenhofen, Germany
- Prof. Peter Fritzon, Linköping University, Sweden
- Dr. Hilding Elmqvist, Dynasim AB, Lund, Sweden
- Dr. Michael Tiller, Emmeskay Inc., Michigan, USA

Program Committee

- Prof. Karl-Erik Årzén, Lund University, Lund, Sweden
- Dr. John Batteh, Emmeskay Inc., Michigan, USA
- Dr. Ingrid Bausch-Gall, Bausch-Gall GmbH, Munich, Germany
- Daniel Bouskela, EDF, Paris, France
- Prof. Felix Breiteneker, University of Technology, Vienna, Austria
- Dr. Thomas Christ, BMW, Michigan, USA
- Prof. Francesco Casella, Politecnico di Milano, Milano, Italy
- Prof. François E. Cellier, ETH Zürich, Zürich, Switzerland
- Mike Dempsey, Claytex Services Limited, Leamington, U.K.
- Denis Fargeton, LMS Imagine, Roanne, France
- Dr. Rüdiger Franke, ABB, Heidelberg, Germany
- Rui Gao, Dassault Systèmes K.K., Nagoya, Japan
- Anton Haumer, Technical Consulting, Vienna, Austria
- Dr. Christian Kral, arsenal research, Vienna, Austria
- Gerard Lecina, Dassault Systèmes, Paris, France
- Dirk Limperich, Daimler AG, Sindelfingen, Germany
- Kilian Link, Siemens AG, Erlangen, Germany
- Dr. Jakob Mauss, QTronic GmbH, Berlin, Germany
- Dr. Ramine Nikoukhah, INRIA, Rocquencourt, France

-
- Franz Pirker, arsenal research, Vienna, Austria
 - Prof. Gerhard Schmitz, Technical University Hamburg-Harburg, Germany
 - Peter Schneider, Fraunhofer IIS/EAS, Dresden, Germany
 - Dr. Edward D. Tate, General Motors, Michigan, USA
 - Dr. Wilhelm Tegethoff, TLK-Thermo GmbH, Braunschweig, Germany
 - Dr. Hubertus Tummescheit, Modelon AB, Lund, Sweden
 - Dr. Andreas Uhlig, ITI GmbH, Dresden, Germany

Local Organizing Committee

- Prof. Bernhard Bachmann
- Dr. Elke Koppenrade
- Jens Schönbohm
- Ralf Derdau
- Eveni, Konferenz-Management-Software, www.eveni.com
- Bielefeld Marketing GmbH, www.bielefeld-marketing.de

Contents

Volume 1	1
Session 1a	
Language, Tools and Algorithms	1
D. Broman, P. Aronsson, P. Fritzson: Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica	3
S. E. Mattsson, H. Elmqvist: Unit Checking and Quantity Conservation	13
H. Olsson, M. Otter, S. E. Mattsson, H. Elmqvist: Balanced Models in Modelica 3.0 for Increased Model Quality	21
Session 1b	
Language, Tools and Algorithms	35
M. Najafi, R. Nikoukhah: Initialization of Modelica Models in Scicos	37
D. Zimmer: Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems	47
J. Åkesson : Optimica—An Extension of Modelica Supporting Dynamic Optimization . .	57
Session 1c	
Automotive Applications	67
J. Batteh, C. Newman: Detailed Simulation of Turbocharged Engines with Modelica . .	69
H. Oberguggenberger, D. Simic: Thermal Modelling of an Automotive Nickel Metall Hybrid Battery in Modelica using Dymola	77
M. Corno, F. Casella, S. M. Savaresi, R. Scattolini: Object Oriented Modeling of a Gasoline Direct Injection System	83
Session 1d	
Electric Systems & Applications	93
M. Kuhn, M. Otter, L. Raulin: A Multi Level Approach for Aircraft Electrical Systems Design	95
C. Schallert : Incorporation of Reliability Analysis Methods with Modelica	103
F. Wagner, L. Liu, G. Frey: Simulation of Distributed Automation Systems in Modelica	113
Session 2a	
Language, Tools and Algorithms	123
A. Jardin, W. Marquis-Favre, D. Thomasset, F. Guillemard, F. Lorenz: Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1	125
T. Johnson, C. Paredis, R. Burkhart: Integrating Models and Simualtions of Continuous Dynamics into SysML	135

A. Leva, F. Donida, M. Bonvini, L. Ravelli: Modelica Library for Logic Control Systems written in the FBD Language	147
Session 2b	
Thermodynamic Systems & Applications	155
F. Casella, C. Richter : ExternalMedia: A Library for Easy Re-Use of External Fluid Property Code in Modelica	157
F. Cellier, J. Greifeneder: ThermoBondLib - A New Modelica Library for Modeling Convective Flows	163
T. Vahlenkamp, S. Wischhusen: FluidDissipation - A Centralised Library for Modelling of Heat Transfer and Pressure Loss	173
Session 2c	
Mechanical Systems & Applications	179
G. Verzichelli : Development of an Aircraft and Landing Gears Model with Steering System in Modelica-Dymola	181
G. Looye : The New DLR Flight Dynamics Library	193
I. Kosenko, E. Alexandrov: Implementation of the Hertz Contact Model and Its Volumetric Modification on Modelica	203
Session 2d	
Electric Systems & Applications	213
D. Winkler, C. Gühmann: Modelling of Electric Drives using freeFOCLib	215
T. Bödrich : Electromagnetic Actuator Modelling with the Extended Modelica Magnetic Library	221
A. Haumer, C. Kral, J. V. Gragger, H. Kapeller: Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors	229
Session 3a	
Language, Tools and Algorithms	237
T. Pulecchi, F. Casella: HyAuLib: Modelling Hybrid Automata in Modelica	239
G. Fish, M. Dempsey : Application of Neural Networks to model Catamaran Type Powerboats	247
M. Malmheden, H. Elmqvist, S. E. Mattsson, D. Henriksson, M. Otter: ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata	255
U. Donath, J. Haufe, T. Blochwitz, T. Neidhold: A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment	269
Session 3b	
Thermodynamic Systems & Applications	277
R. Franke, B.S. Babij, M. Antoine, A. Isaksson: Model-Based Online Applications in the ABB Dynamic Optimization Framework	279
J. I. Videla, B. Lie: Using Modelica/Matlab for Parameter Estimation in a Bioethanol Fermentation Model	287
L. Imsland, P. Kittilsen, T. Steinar Schei: Model-Based Optimizing Control and Estimation using Modelica Models	301
F. Casella, F. Donida, B. Bachmann, P. Aronsson: Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models	311

Session 3c	
Automotive Applications	319
W. Chen, G. Qin, L. Li, Y. Zhang, L. Chen: Modelling of Conventional Vehicle in Modelica	321
J. Andreasson, M. Jonasson: Vehicle Model for Limit Handling: Implementation and Validation	327
H. Isernhagen, C. Gühmann: Modelling of a Double Clutch Transmission with an Appropriate Controller for the Simulation of Shifting Processes	333
A. Junghanns, J. Mauss, M. Tatar: TestWeaver - A Tool for Simulation-Based Test of Mechatronic Designs	341
Session 3d	
Electric Systems & Applications	349
C. Kral, A. Haumer: Simulation of Electrical Rotor Asymmetries in Squirrel Cage Induction Machines with the ExtendedMachines Library	351
H. Kapeller, A. Haumer, C. Kral, G. Pascoli, F. Pirker: Modeling and Simulation of a Large Chipper Drive	361
H. Giuliani, C. J. Fenz, A. Haumer, H. Kapeller: Simulation and Validation of Power Losses in the Buck-Converter Model included in the SmartElectricDrives Library . . .	369
A. Ebner, M. Ganchev, H. Oberguggenberger, F. Pirker: Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC	375
 Volume 2	 381
Session 4a	
Language, Tools and Algorithms	381
A. Abel, T. Nähring: Frequency-Domain Analysis Methods for Modelica Models	383
F. Cellier : World3 in Modelica: Creating System Dynamics Models in the Modelica Framework	393
F. Donida, A. Leva: Modelica as a Host Language for Process/Control Co-Simulation and Co-Design	401
A. Pop, K. Stavåker, P. Fritzson: Exception Handling for Modelica	409
Session 4b	
Thermodynamic Systems & Applications	419
J. Fahlke, S. Püschel, F. Hannemann, B. Meyer: Modelling of the Gasification Island with Modelica	421
M. Bockholt, W. Tegethoff, N. Lemke, N.-C. Strupp, C. Richter: Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles	429
C. Junior, C. Richter, W. Tegethoff, N. Lemke, J. Köhler: Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
P. Li, Y. Li, J. Seem: Dynamic Modeling and Self-Optimizing Control of Air-Side Economizers	447
Session 4c	
Automotive Applications	465

M. Najafi, Z. Benjelloun-Dabaghi: Using Modelica for Modeling and Simulation of Spark Ignited Engine and Drilling Station in IFP	467
S. Karim, H. Tummescheit: Controller Development for an Automotive Ac-system using R744 as Refrigerant	477
H. Wigermo, J. von Grundherr, T. Christ: Implementation of a Modelica Online Optimization for an Operating Strategy of a Hybrid Powertrain	487
E. Tate, M. Sasena, J. Gohl, M. Tiller: Model Embedded Control: A Method to Rapidly Synthesize Controllers in a Modeling Environment	493

Session 4d

Mechanical Systems & Applications	503
F. Casella, M. Lovera: High-Accuracy Orbital Dynamics Simulation through Keplerian and Equinoctial Parameters	505
J. Andreasson, M. Gäfvert: Rotational3D - Efficient Modelling of 3D Effects in Rotational Mechanics	515
S. Wolf, J. Haase, C. Clauß, M. Jöckel, J. Lösch: Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer Bushings	521
M. Pfennig, F. Thielecke: Implementation of a Modelica Library for Simulation of High-Lift Drive Systems	531

Session 5

Poster Session	541
T. Hirsch, M. Eck: 4-Dimensional Table Interpolation with Modelica	543
M. Höbinger, M. Otter : PlanarMultiBody - A Modelica Library for Planar Multi-Body Systems	549
D. Simic, T. Bäuml: Implementation of Hybrid Electric Vehicles using the VehicleInterfaces and the SmartElectricDrives Libraries	557
P. Machanick, A. Liebman, P. Fritzson: Modeling of CO2 Reduction Impacts on Energy Prices with Modelica	565
M. Schicktanz : Modelling of an Adsorption Chiller with Modelica	573
T. Blochwitz, G. Kurzbach, T. Neidhold : An External Model Interface for Modelica	579
B. El Hefni, B. Bride, B. Pechine: Two Steady State CHP Models with Modelica : Mirafiori overall Model and Multi-configuration Biomass Model	585
J. V. Gragger, A. Haumer, C. Kral, F. Pirker: Efficient Analysis of Harmonic Losses in PWM Voltage Source Induction Machine Drives with Modelica	593
J. Haase, S. Wolf, C. Clauß: Monte Carlo Simulation with Modelica	601
O. Enge-Rosenblatt, C. Clauß, P. Schwarz, F. Breitenecker, C. Nytsch-Geusen: Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605
O. Enge-Rosenblatt, P. Schneider: Modelica Wind Turbine Models with Structural Changes Related to Different Operating Modes	611
K. Tuszyński : ExcelInterface - A Tool for Interfacing Dymola through Excel	621
K. Dietl, J. Vasel, G. Schmitz, W. Casas, C. Mehrkens: Modeling of Cold Plates for Power Electronic Cooling	627
N. Philipson, J. Andreasson, M. Gäfvert, A. Woodruff: Heavy Vehicle Modeling with VehicleDynamics Library	629

Session 6a

Language, Tools and Algorithms	635
K. Stavåker, A. Pop, P. Fritzson: Compiling and Using Pattern Matching in Modelica	637
M. Tiller : Patterns and Anti-Patterns in Modelica	647
P. Fritzson, A. Pop, K. Norling, M. Blom: Comment- and Indentation Preserving Refactoring and Unparsing for Modelica	657

Session 6b

Language, Tools and Algorithms	667
A. Elsheikh, S. Noack, W. Wiechert: Sensitivity Analysis of Modelica Applications via Automatic Differentiation	669
R. Nikoukhah, S. Furic : Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model	677
F. Dshabarow, F. Cellier, D. Zimmer: Support for Dymola in the Modeling and Simulation of Physical Systems with Distributed Parameters	683

Session 6c

Thermodynamic Systems & Applications	691
H. Tummescheit, K. Tuszynski, P. Arnold: Simulation of Peak Stresses and Bowing Phenomena during the Cool Down of a Cryogenic Transfer System	693
A. Joos, G. Schmitz, W. Casas: Enhancement of a Modelica Model of a Desiccant Wheel	701
M. Gäfvert, T. Skoglund, H. Tummescheit, J. Windahl, H. Wikander, P. Reuterswård: Real-Time HWIL Simulation of Liquid Food Process Lines	709

Session 6d

Mechanical Systems & Applications	717
T. Juhász, U. Schmucker: Automatic Model Conversion to Modelica for Dymola-based Mechatronic Simulation	719
I. I. Kosenko, A. S. Kuleshov : Modelica Implementation of the Skateboard Dynamics	727
T. Hoefft, C. Nytsch-Geusen: Design and Validation of an Annotation-Concept for the Representation of 3D-Geometries in Modelica	735



Index of Authors

Åkesson, Johan: <i>Lund University, Lund, Sweden</i> Optimica—An Extension of Modelica Supporting Dynamic Optimization	57
Abel, Andreas: <i>ITI GmbH, Dresden, Germany</i> Frequency-Domain Analysis Methods for Modelica Models	383
Alexandrov, Evgeniy: <i>Moscow State University of Tourism and Service, Moscow, Russian Federation</i> Implementation of the Hertz Contact Model and Its Volumetric Modification on Modelica	203
Andreasson, Johan: <i>Modelon AB, Lund, Sweden</i> Heavy Vehicle Modeling with VehicleDynamics Library	629
Rotational3D - Efficient Modelling of 3D Effects in Rotational Mechanics	515
Vehicle Model for Limit Handling: Implementation and Validation	327
Antoine, Marc: <i>ABB Power Technology Systems, Mannheim, Germany</i> Model-Based Online Applications in the ABB Dynamic Optimization Framework	279
Arnold, Philip: <i>Linde Kryotechik AG, Pfungen, Switzerland</i> Simulation of Peak Stresses and Bowing Phenomena during the Cool Down of a Cryogenic Transfer System	693
Aronsson, Peter: <i>Mathcore Engeneering, Linköping, Sweden</i> Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica	3
Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models	311
Bäumel, Thomas: <i>Arsenal Research, Vienna, Austria</i> Implementation of Hybrid Electric Vehicles using the VehicleInterfaces and the SmartElectricDrives Libraries	557
Bödrich, Thomas: <i>Dresden University of Technology, Dresden, Germany</i> Electromagnetic Actuator Modelling with the Extended Modelica Magnetic Library	221
Babji B.S.: <i>ABB Corporate Research, Bangalore, India</i> Model-Based Online Applications in the ABB Dynamic Optimization Framework	279
Bachmann, Bernhard: <i>Bielefeld University of Applied Sciences, Bielefeld, Germany</i> Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models	311
Batteh, John: <i>Ford Motor Company, Dearborn, U.S.A.</i> Detailed Simulation of Turbocharged Engines with Modelica	69
Benjelloun-Dabaghi, Zakia: <i>INRIA, Rocquencourt, France</i> Using Modelica for Modeling and Simulation of Spark Ignited Engine and Drilling Station in IFP	467
Blochwitz, Torsten: <i>ITI GmbH, Dresden, Germany</i>	

A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment.....	269
An External Model Interface for Modelica	579
Blom, Mikael: <i>Linköping University, Linköping, Sweden</i>	
Comment- and Indentation Preserving Refactoring and Unparsing for Modelica	657
Bockholt, Marcos: <i>Braunschweig University of Technology, Braunschweig, Germany</i>	
Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles.....	429
Bonvini, Marco: <i>Politecnico di Milano, Milano, Italy</i>	
Modelica Library for Logic Control Systems written in the FBD Language	147
Breitenecker, Felix: <i>Vienna University of Technology, Vienna, Austria</i>	
Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605
Bride, Benoît: <i>EDF R&D, Chatou, France</i>	
Two Steady State CHP Models with Modelica: Mirafiori overall Model and Multi-configuration Biomass Model	585
Broman, David: <i>Linköping University, Linköping, Sweden</i>	
Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica ..	3
Burghart, Roger: <i>Hamburg University of Technology, Hamburg, Germany</i>	
Integrating Models and Simulations of Continuous Dynamics into SysML	135
Casas, Wilson: <i>Hamburg University of Technology, Hamburg, Germany</i>	
Enhancement of a Modelica Model of a Desiccant Wheel.....	701
Casas, Wilson: <i>Hamburg-Harburg University of Technology, Hamburg, Germany</i>	
Modeling of Cold Plates for Power Electronic Cooling.....	627
Casella, Francesco: <i>Politecnico di Milano, Milano, Italy</i>	
ExternalMedia: A Library for Easy Re-Use of External Fluid Property Code in Modelica....	157
High-Accuracy Orbital Dynamics Simulation through Keplerian and Equinoctial Parameters	505
HyAuLib: Modelling Hybrid Automata in Modelica.....	239
Object Oriented Modeling of a Gasoline Direct Injection System	83
Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models	311
Cellier, François: <i>ETH Zürich, Zürich, Switzerland</i>	
Support for Dymola in the Modeling and Simulation of Physical Systems with Distributed Parameters.....	683
ThermoBondLib - A New Modelica Library for Modeling Convective Flows	163
World3 in Modelica: Creating System Dynamics Models in the Modelica Framework	393
Chen, Liping: <i>Huazhong University of Science and Technology, Wuhan, China</i>	
Modelling of Conventional Vehicle in Modelica	321
Chen, Wei: <i>Huazhong University of Science and Technology, Wuhan, China</i>	
Modelling of Conventional Vehicle in Modelica	321
Christ, Thomas: <i>BMW Hybrid Cooperation, Troy, U.S.A.</i>	
Implementation of a Modelica Online Optimization for an Operating Strategy of a Hybrid Powertrain	487
Clauß, Christoph: <i>Fraunhofer Institut, Dresden, Germany</i>	
Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605
Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer Bushings	521

Monte Carlo Simulation with Modelica.....	601
Corno, Matteo: <i>Politecnico di Milano, Milano, Italy</i>	
Object Oriented Modeling of a Gasoline Direct Injection System	83
Dempsey, Mike: <i>Claytex Services Ltd, Leamington Spa, United Kingdom</i>	
Application of Neural Networks to model Catamaran Type Powerboats	247
Dietl, Karin: <i>Hamburg-Harburg University of Technology, Hamburg, Germany</i>	
Modeling of Cold Plates for Power Electronic Cooling.....	627
Donath, Ulrich: <i>Fraunhofer Institut, Dresden, Germany</i>	
A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment.....	269
Donida, Filippo: <i>Politecnico di Milano, Milano, Italy</i>	
Modelica as a Host Language for Process/Control Co-Simulation and Co-Design	401
Modelica Library for Logic Control Systems written in the FBD Language	147
Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models	311
Dshabarow, Farid: <i>ABB Turbo Systems AG, Baden, Switzerland</i>	
Support for Dymola in the Modeling and Simulation of Physical Systems with Distributed Parameters.....	683
Ebner, Arno: <i>Arsenal Research, Vienna, Austria</i>	
Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC.....	375
Eck, Markus: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i>	
4-Dimensional Table Interpolation with Modelica	543
El Hefni, Baligh: <i>EDF R&D, Chatou, France</i>	
Two Steady State CHP Models with Modelica: Mirafiori overall Model and Multi-configuration Biomass Model	585
Elmqvist, Hilding: <i>Dynasim AB, Lund, Sweden</i>	
Balanced Models in Modelica 3.0 for Increased Model Quality	21
ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata.....	255
Unit Checking and Quantity Conservation	13
Elsheikh, Atya: <i>Siegen University, Siegen, Germany</i>	
Sensitivity Analysis of Modelica Applications via Automatic Differentiation	669
Enge-Rosenblatt, Olaf: <i>Fraunhofer Institut, Dresden, Germany</i>	
Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605
Modelica Wind Turbine Models with Structural Changes Related to Different Operating Modes.....	611
Fahlke, Julia: <i>Freiberg University of Technology, Freiberg, Germany</i>	
Modelling of the Gasification Island with Modelica	421
Fenz, Claus J. : <i>Arsenal Research, Vienna, Austria</i>	
Simulation and Validation of Power Losses in the Buck-Converter Model included in the SmartElectricDrives Library	369
Fish, Garron: <i>Claytex Services Ltd, Leamington Spa, United Kingdom</i>	
Application of Neural Networks to model Catamaran Type Powerboats	247
Franke, Ruediger: <i>ABB Power Technology Systems, Mannheim, Germany</i>	
Model-Based Online Applications in the ABB Dynamic Optimization Framework	279
Frey, Georg: <i>Kaiserslautern University of Technology, Kaiserslautern, Germany</i>	

Simulation of Distributed Automation Systems in Modelica	113
Fritzson, Peter: <i>Linköping University, Linköping, Sweden</i>	
Comment- and Indentation Preserving Refactoring and Unparsing for Modelica	657
Compiling and Using Pattern Matching in Modelica	637
Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica ..	3
Exception Handling for Modelica	409
Modeling of CO2 Reduction Impacts on Energy Prices with Modelica	565
Furic, Sébastien: <i>LMS-Imagine, Roanne, France</i>	
Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid	
Model	677
Gäfvert, Magnus: <i>Modelon AB, Lund, Sweden</i>	
Heavy Vehicle Modeling with VehicleDynamics Library	629
Real-Time HWIL Simulation of Liquid Food Process Lines	709
Rotational3D - Efficient Modelling of 3D Effects in Rotational Mechanics	515
Gühmann, Clemens: <i>Technische Universität Berlin, Berlin, Germany</i>	
Modelling of a Double Clutch Transmission with an Appropriate Controller for the	
Simulation of Shifting Processes	333
Modelling of Electric Drives using freeFOCLib	215
Ganchev, Martin: <i>Arsenal Research, Vienna, Austria</i>	
Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC	375
Giuliani, Harald: <i>Arsenal Research, Vienna, Austria</i>	
Simulation and Validation of Power Losses in the Buck-Converter Model included in the	
SmartElectricDrives Library	369
Gohl, Jesse: <i>Emmeskay, Inc., Plymouth, U.S.A.</i>	
Model Embedded Control: A Methode to Rapidly Synthesize Controllers in a Modeling	
Environment	493
Gragger, Johannes V.: <i>Arsenal Research, Vienna, Austria</i>	
Efficient Analysis of Harmonic Losses in PWM Voltage Source Induction Machine Drives	
with Modelica	593
Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors ...	229
Greifeneder, Jürgen: <i>Kaiserslautern University of Technology, Kaiserslautern, Germany</i>	
ThermoBondLib - A New Modelica Library for Modeling Convective Flows	163
Guillemard, Franck: <i>PSA Peugeot Citroën, Vélizy-Villacoublay Cedex, France</i>	
Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1125	
Höbinger, Mathias: <i>Vienna University of Technology, Vienna, Austria</i>	
PlanarMultiBody - A Modelica Library for Planar Multi-Body Systems	549
Haase, Joachim: <i>Fraunhofer Institut, Dresden, Germany</i>	
Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer	
Bushings	521
Monte Carlo Simulation with Modelica	601
Hannemann, Frank: <i>Siemens Fuel Gasification Technologie, Freiberg, Germany</i>	
Modelling of the Gasification Island with Modelica	421
Haufe, Jürgen: <i>Fraunhofer Institut, Dresden, Germany</i>	
A new Approach for Modeling and Verification of Discrete Control Components within	
a Modelica Environment	269
Haumer, Anton: <i>Arsenal Research, Vienna, Austria</i>	

Efficient Analysis of Harmonic Losses in PWM Voltage Source Induction Machine Drives with Modelica	593
Modeling and Simulation of a Large Chipper Drive	361
Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors	229
Simulation and Validation of Power Losses in the Buck-Converter Model included in the SmartElectricDrives Library	369
Simulation of Electrical Rotor Asymmetries in Squirrel Cage Induction Machines with the ExtendedMachines Library	351
Henriksson, Dan: <i>Dynasim AB, Lund, Sweden</i> ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata.....	255
Hirsch, Tobias: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i> 4-Dimensional Table Interpolation with Modelica	543
Hoeft, Thomas: <i>Fraunhofer Institut, Berlin, Germany</i> Design and Validation of an Annotation-Concept for the Representation of 3D-Geometries in Modelica	735
Imsland, Lars: <i>Cybernetica AS, Trondheim, Norway</i> Model-Based Optimizing Control and Estimation using Modelica Models.....	301
Isaksson, Alf: <i>ABB Corporate Research, Bangalore, India</i> Model-Based Online Applications in the ABB Dynamic Optimization Framework	279
Isernhagen, Henrik: <i>Technische Universität Berlin, Berlin, Germany</i> Modelling of a Double Clutch Transmission with an Appropriate Controller for the Simulation of Shifting Processes	333
Jöckel, Michael: <i>Fraunhofer Institut, Darmstadt, Germany</i> Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer Bushings	521
Jardin, Audrey: <i>INSA-Lyon AMPERE, Villeurbanne Cedex, France</i> Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1125	
Johnson, Thomas: <i>Georgia Institute of Technology, Atlanta, U.S.A.</i> Integrating Models and Simulations of Continuous Dynamics into SysML	135
Jonasson, Mats: <i>Volvo Car Corporation, Göteborg, Sweden</i> Vehicle Model for Limit Handling: Implementation and Validation	327
Joos, Andreas: <i>Hamburg University of Technology, Hamburg, Germany</i> Enhancement of a Modelica Model of a Desiccant Wheel	701
Juhász, Tamás: <i>Fraunhofer Institut, Magdeburg, Germany</i> Automatic Model Conversion to Modelica for Dymola-based Mechatronic Simulation.....	719
Junghanns, Andreas: <i>QTronic GmbH, Berlin, Germany</i> TestWeaver - A Tool for Simulation-Based Test of Mechatronic Designs	341
Junior, Christine: <i>Braunschweig University of Technology, Braunschweig, Germany</i> Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
Köhler, Jürgen: <i>Braunschweig University of Technology, Braunschweig, Germany</i> Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
Kapeller, Hansjörg: <i>Arsenal Research, Vienna, Austria</i> Modeling and Simulation of a Large Chipper Drive	361

Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors	229
Simulation and Validation of Power Losses in the Buck-Converter Model included in the SmartElectricDrives Library	369
Karim, Sanaz: <i>Modelon AB, Lund, Sweden</i>	
Controller Development for an Automotive Ac-system using R744 as Refrigerant	477
Kittilsen, Pål: <i>Cybernetica AS, Trondheim, Norway</i>	
Model-Based Optimizing Control and Estimation using Modelica Models	301
Kosenko, Ivan I.: <i>Moscow State University of Tourism and Service, Moscow, Russian Federation</i>	
Implementation of the Hertz Contact Model and Its Volumetric Modification on Modelica	203
Modelica Implementation of the Skateboard Dynamics	727
Kral, Christian: <i>Arsenal Research, Vienna, Austria</i>	
Efficient Analysis of Harmonic Losses in PWM Voltage Source Induction Machine Drives with Modelica	593
Modeling and Simulation of a Large Chipper Drive	361
Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors	229
Simulation of Electrical Rotor Asymmetries in Squirrel Cage Induction Machines with the ExtendedMachines Library	351
Kuhn, Martin: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i>	
A Multi Level Approach for Aircraft Electrical Systems Design	95
Kuleshov, Alexander S.: <i>Lomonosov Moscow State University, Moscow, Russian Federation</i>	
Modelica Implementation of the Skateboard Dynamics	727
Kurzbach, Gerd: <i>ITI GmbH, Dresden, Germany</i>	
An External Model Interface for Modelica	579
Lösch, Jürgen: <i>Fraunhofer Institut, Darmstadt, Germany</i>	
Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer Bushings	521
Lemke, Nicholas: <i>Braunschweig University of Technology, Braunschweig, Germany</i>	
Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
Lemke, Nicholas: <i>TLK-Thermo GmbH, Braunschweig, Germany</i>	
Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles	429
Leva, Alberto: <i>Politecnico di Milano, Milano, Italy</i>	
Modelica as a Host Language for Process/Control Co-Simulation and Co-Design	401
Modelica Library for Logic Control Systems written in the FBD Language	147
Li, Lingyang: <i>Huazhong University of Science and Technology, Wuhan, China</i>	
Modelling of Conventional Vehicle in Modelica	321
Li, Pengfei: <i>University of Wisconsin, Milwaukee, U.S.A.</i>	
Dynamic Modeling and Self-Optimizing Control of Air-Side Economizers	447
Li, Yaoyu: <i>University of Wisconsin, Milwaukee, U.S.A.</i>	
Dynamic Modeling and Self-Optimizing Control of Air-Side Economizers	447
Lie, Bernt: <i>Telemark University College, Porsgrunn, Norway</i>	
Using Modelica/Matlab for Parameter Estimation in a Bioethanol Fermentation Model	287
Liebman, Ariel: <i>University of Queensland, Brisbane, Australia</i>	
Modeling of CO2 Reduction Impacts on Energy Prices with Modelica	565
Liu, Liu: <i>Kaiserslautern University of Technology, Kaiserslautern, Germany</i>	

Simulation of Distributed Automation Systems in Modelica	113
Looye, Gertjan: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i>	
The New DLR Flight Dynamics Library	193
Lorenz, Francis: <i>LorSim, Liège, France</i>	
Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1125	
Lovera, Marco: <i>Politecnico di Milano, Milano, Italy</i>	
High-Accuracy Orbital Dynamics Simulation through Keplerian and Equinoctial Parameters	505
Machanick, Philip: <i>University of Queensland, Brisbane, Australia</i>	
Modeling of CO2 Reduction Impacts on Energy Prices with Modelica	565
Malmheden, Martin: <i>Dynasim AB, Lund, Sweden</i>	
ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata	255
Marquis-Favre, Wilfrid: <i>INSA-Lyon AMPERE, Villeurbanne Cedex, France</i>	
Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1125	
Mattsson, Sven Erik: <i>Dynasim AB, Lund, Sweden</i>	
Balanced Models in Modelica 3.0 for Increased Model Quality	21
ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata	255
Unit Checking and Quantity Conservation	13
Mauss, Jakob: <i>QTronic GmbH, Berlin, Germany</i>	
TestWeaver - A Tool for Simulation-Based Test of Mechatronic Designs	341
Mehrkens, Christian: <i>Hamburg-Harburg University of Technology, Hamburg, Germany</i>	
Modeling of Cold Plates for Power Electronic Cooling	627
Meyer, Bernd: <i>Freiberg University of Technology, Freiberg, Germany</i>	
Modelling of the Gasification Island with Modelica	421
Nähring, Tobias: <i>ITI GmbH, Dresden, Germany</i>	
Frequency-Domain Analysis Methods for Modelica Models	383
Najafi, Masoud: <i>INRIA, Rocquencourt, France</i>	
Initialization of Modelica Models in Scicos	37
Using Modelica for Modeling and Simulation of Spark Ignited Engine and Drilling Station in IFP	467
Neidhold, Thomas: <i>ITI GmbH, Dresden, Germany</i>	
A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment	269
An External Model Interface for Modelica	579
Newman, Charles: <i>Ford Motor Company, Dearborn, U.S.A.</i>	
Detailed Simulation of Turbocharged Engines with Modelica	69
Nikoukhah, Ramine: <i>INRIA, Rocquencourt, France</i>	
Initialization of Modelica Models in Scicos	37
Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model	677
Noack, Stephan: <i>Research Center Jülich GmbH, Jülich, Germany</i>	
Sensitivity Analysis of Modelica Applications via Automatic Differentiation	669
Norling, Kristoffer: <i>Linköping University, Linköping, Sweden</i>	
Comment- and Indentation Preserving Refactoring and Unparsing for Modelica	657
Nytsch-Geusen, Christoph: <i>Fraunhofer Institut, Berlin, Germany</i>	
Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605

Design and Validation of an Annotation-Concept for the Representation of 3D-Geometries in Modelica	735
Oberguggenberger, Helmut: <i>Arsenal Research, Vienna, Austria</i>	
Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC	375
Thermal Modelling of an Automotive Nickel Metall Hydrid Battery in Modelica using Dymola	77
Olsson, Hans: <i>Dynasim AB, Lund, Sweden</i>	
Balanced Models in Modelica 3.0 for Increased Model Quality	21
Otter, Martin: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i>	
A Multi Level Approach for Aircraft Electrical Systems Design	95
Balanced Models in Modelica 3.0 for Increased Model Quality	21
ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata	255
PlanarMultiBody - A Modelica Library for Planar Multi-Body Systems	549
Püschel, Stephan: <i>Freiberg University of Technology, Freiberg, Germany</i>	
Modelling of the Gasification Island with Modelica	421
Paredis, Chris: <i>Georgia Institute of Technology, Atlanta, U.S.A.</i>	
Integrating Models and Simualtions of Continuous Dynamics into SysML	135
Pascoli, Gert: <i>Arsenal Research, Vienna, Austria</i>	
Modeling and Simulation of a Large Chipper Drive	361
Pechine, Bruno: <i>EDF R&D, Chatou, France</i>	
Two Steady State CHP Models with Modelica: Mirafiori overall Model and Multi-configuration Biomass Model	585
Pfennig, Malte: <i>Hamburg University of Technology, Hamburg, Germany</i>	
Implementation of a Modelica Library for Simulation of High-Lift Drive Systems	531
Philipson, Niklas: <i>Modelon AB, Lund, Sweden</i>	
Heavy Vehicle Modeling with VehicleDynamics Library	629
Pirker, Franz: <i>Arsenal Research, Vienna, Austria</i>	
Efficient Analysis of Harmonic Losses in PWM Voltage Source Induction Machine Drives with Modelica	593
Modeling and Simulation of a Large Chipper Drive	361
Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC	375
Pop, Adrian: <i>Linköping University, Linköping, Sweden</i>	
Comment- and Indentation Preserving Refactoring and Unparsing for Modelica	657
Compiling and Using Pattern Matching in Modelica	637
Exception Handling for Modelica	409
Pulecchi, Tiziano: <i>Politecnico di Milano, Milano, Italy</i>	
HyAuLib: Modelling Hybrid Automata in Modelica	239
Qin, Gang: <i>Huazhong University of Science and Technology, Wuhan, China</i>	
Modelling of Conventional Vehicle in Modelica	321
Raulin, Loic: <i>Airbus, Toulouse, France</i>	
A Multi Level Approach for Aircraft Electrical Systems Design	95
Ravelli, Lorenzo: <i>Politecnico di Milano, Milano, Italy</i>	
Modelica Library for Logic Control Systems written in the FBD Language	147
Reuterswärd, Philip: <i>Modelon AB, Lund, Sweden</i>	

Real-Time HWIL Simulation of Liquid Food Process Lines	709
Richter, Christoph: <i>Braunschweig University of Technology, Braunschweig, Germany</i>	
ExternalMedia: A Library for Easy Re-Use of External Fluid Property Code in Modelica....	157
Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles	429
Sasena, Michael: <i>Emmeskay, Inc., Plymouth, U.S.A.</i>	
Model Embedded Control: A Methode to Rapidly Synthesize Controllers in a Modeling Environment	493
Savaresi, Sergio M.: <i>Politecnico di Milano, Milano, Italy</i>	
Object Oriented Modeling of a Gasoline Direct Injection System	83
Scattolini, Riccardo: <i>Politecnico di Milano, Milano, Italy</i>	
Object Oriented Modeling of a Gasoline Direct Injection System	83
Schallert, Christian: <i>German Aerospace Center, Oberpfaffenhofen, Germany</i>	
Incorporation of Reliability Analysis Methods with Modelica	103
Schei, Tor Steinar: <i>Cybernetica AS, Trondheim, Norway</i>	
Model-Based Optimizing Control and Estimation using Modelica Models	301
Schicktanz, Matthias: <i>Fraunhofer Institut, Freiburg, Germany</i>	
Modelling of an Adsorption Chiller with Modelica	573
Schmitz, Gerhard: <i>Hamburg University of Technology, Hamburg, Germany</i>	
Enhancement of a Modelica Model of a Desiccant Wheel	701
Schmitz, Gerhard: <i>Hamburg-Harburg University of Technology, Hamburg, Germany</i>	
Modeling of Cold Plates for Power Electronic Cooling	627
Schmucker, Ulrich: <i>Fraunhofer Institut, Magdeburg, Germany</i>	
Automatic Model Conversion to Modelica for Dymola-based Mechatronic Simulation	719
Schneider, Peter: <i>Fraunhofer Institut, Dresden, Germany</i>	
Modelica Wind Turbine Models with Structural Changes Related to Different Operating Modes	611
Schwarz, Peter: <i>Fraunhofer Institut, Dresden, Germany</i>	
Comparisons of Different Modelica-Based Simulators Using Benchmark Tasks	605
Seem, John: <i>Building Efficiency Research Group, Milwaukee, U.S.A.</i>	
Dynamic Modeling and Self-Optimizing Control of Air-Side Economizers	447
Simic, Dragan: <i>Arsenal Research, Vienna, Austria</i>	
Implementation of Hybrid Electric Vehicles using the VehicleInterfaces and the SmartElectricDrives Libraries	557
Thermal Modelling of an Automotive Nickel Metall Hydrid Battery in Modelica using Dymola	77
Skoglund, Tomas: <i>Tetra Pak Procesing Systems, Lund, Sweden</i>	
Real-Time HWIL Simulation of Liquid Food Process Lines	709
Stavåker, Kristian: <i>Linköping University, Linköping, Sweden</i>	
Compiling and Using Pattern Matching in Modelica	637
Exception Handling for Modelica	409
Strupp, Nils-Christian: <i>Braunschweig University of Technology, Braunschweig, Germany</i>	
Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles	429

Tatar, Muger: <i>QTronic GmbH, Berlin, Germany</i>	
TestWeaver - A Tool for Simulation-Based Test of Mechatronic Designs	341
Tate, Edward: <i>General Motors, Michigan, U.S.A.</i>	
Model Embedded Control: A Methode to Rapidly Synthesize Controllers in a Modeling Environment	493
Tegethoff, Wilhelm: <i>Braunschweig University of Technology, Braunschweig, Germany</i>	
Modeling and Simulation of a Thermoelectric Heat Exchanger using the Object-Oriented Library TIL	437
Transient Modelling of a Controllable Low Pressure Accumulator in CO2 Refrigeration Cycles	429
Thielecke, Frank: <i>Hamburg University of Technology, Hamburg, Germany</i>	
Implementation of a Modelica Library for Simulation of High-Lift Drive Systems	531
Thomasset, Daniel: <i>INSA-Lyon AMPERE, Villeurbanne Cedex, France</i>	
Study of a Sizing Methodology and a Modelica Code Generator for the Bond Graph Tool MS1125	
Tiller, Michael: <i>Emmeskay, Inc., Plymouth, U.S.A.</i>	
Model Embedded Control: A Methode to Rapidly Synthesize Controllers in a Modeling Environment	493
Patterns and Anti-Patterns in Modelica	647
Tummescheit, Hubertus: <i>Modelon AB, Lund, Sweden</i>	
Controller Development for an Automotive Ac-system using R744 as Refrigerant	477
Real-Time HWIL Simulation of Liquid Food Process Lines	709
Simulation of Peak Stresses and Bowing Phenomena during the Cool Down of a Cryogenic Transfer System	693
Tuszynski, Kristian: <i>Modelon AB, Lund, Sweden</i>	
ExcelInterface - A Tool for Interfacing Dymola through Excel	621
Simulation of Peak Stresses and Bowing Phenomena during the Cool Down of a Cryogenic Transfer System	693
Vahlenkamp, Thorben: <i>XRG Simulation GmbH, Hamburg, Germany</i>	
FluidDissipation - A Centralised Library for Modelling of Heat Transfer and Pressure Loss ..	173
Vasel, Jens: <i>Hamburg-Harburg University of Technology, Hamburg, Germany</i>	
Modeling of Cold Plates for Power Electronic Cooling	627
Verzichelli, Gianluca: <i>Airbus, Filton, United Kingdom</i>	
Development of an Aircraft and Landing Gears Model with Steering System in Modelica-Dymola	181
Videla, Juan Ignacio: <i>Telemark University College, Porsgrunn, Norway</i>	
Using Modelica/Matlab for Parameter Estimation in a Bioethanol Fermentation Model	287
von Grundherr, Johannes: <i>BMW Group, Munic, Germany</i>	
Implementation of a Modelica Online Optimization for an Operating Strategy of a Hybrid Powertrain	487
Wagner, Florian: <i>Kaiserslautern University of Technology, Kaiserslautern, Germany</i>	
Simulation of Distributed Automation Systems in Modelica	113
Wiechert, Wolfgang: <i>Siegen University, Siegen, Germany</i>	
Sensitivity Analysis of Modelica Applications via Automatic Differentiation	669
Wigermo, Henrik: <i>BMW Group, Munic, Germany</i>	
Implementation of a Modelica Online Optimization for an Operating Strategy of a Hybrid Powertrain	487

Wikander, Hans: <i>Avensia Innovation AB, Lund, Sweden</i>	
Real-Time HWIL Simulation of Liquid Food Process Lines	709
Windahl, Johan: <i>Modelon AB, Lund, Sweden</i>	
Real-Time HWIL Simulation of Liquid Food Process Lines	709
Winkler, Dietmar: <i>Technische Universität Berlin, Berlin, Germany</i>	
Modelling of Electric Drives using freeFOClib	215
Wischhusen, Stefan: <i>XRG Simulation GmbH, Hamburg, Germany</i>	
FluidDissipation - A Centralised Library for Modelling of Heat Transfer and Pressure Loss ..	173
Wolf, Susann: <i>Fraunhofer Institut, Dresden, Germany</i>	
Methods of Sensitivity Calculation Applied to a Multi-Axial Test Rig for Elastomer	
Bushings	521
Monte Carlo Simulation with Modelica.....	601
Woodruff, Andrew: <i>Modelon AB, Lund, Sweden</i>	
Heavy Vehicle Modeling with VehicleDynamics Library	629
Zhang, Yunqing: <i>Huazhong University of Science and Technology, Wuhan, China</i>	
Modelling of Conventional Vehicle in Modelica	321
Zimmer, Dirk: <i>ETH Zürich, Zürich, Switzerland</i>	
Introducing Sol: A General Methodology for Equation-Based Modeling of	
Variable-Structure Systems	47
Support for Dymola in the Modeling and Simulation of Physical Systems with	
Distributed Parameters	683

Session 1a

Language, Tools and Algorithms

Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica

David Broman¹ Peter Aronsson² Peter Fritzson¹

¹Department of Computer and Information Science,
Linköping University, Sweden, {davbr,petfr}@ida.liu.se

²MathCore Engineering, Sweden, peter.aronsson@mathcore.com

Abstract

The Modelica language supports syntax for declaring physical units of variables, but it does not yet exist any defined semantics for how dimensional and unit consistency checking should be carried out. In this paper we explore different approaches and new constructs for improved dimensional inference and unit consistency checking in Modelica; both from an end-user, library, and tool perspective. A proposal for how dimensional inference and unit checking can be carried out is outlined and a prototype implementation is developed and verified using several examples from the Modelica standard library.

Keywords: dimensional analysis, unit checking; dimensions; types; Modelica; language design

1 Introduction

The Modelica language enables expressive modeling by making use of object-oriented acausal constructs. However, certain powerful language constructs easily lead to modeling errors, which are often hard to detect at simulation time. One class of modeling errors that can be detected statically before simulation is model and equation consistency with regards to physical *dimensions*, *quantities* and *units*. The Modelica language specification [12] states how units and quantities can be declared. However, the semantics and strategy for how physical units and dimension of quantities can be checked for consistency, are not described in the specification.

Several of the available tools (e.g., Dymola[4] and Simulation X[8]) implement various algorithms for handling units and dimensions. Furthermore, tool specific language constructs are being added to enable

better unit consistency checking. However, this may lead to incompatibility, where some tools reject certain model and others accept them. Unit related research results within the field of programming language (e.g., [1, 5, 9, 13, 18]) have shown that there exist many concepts and constructs that affect the possibility and simplicity to perform correct dimensional and unit checking. Design considerations must be taken from both the *end user perspective* and from the *library and tool implementor perspective*.

This paper introduces and discusses several different concepts and constructs, which are important when designing a language with support for dimensional inference and unit consistency checking¹. Examples are given using both existing Modelica syntax, and additional suggested constructs. The main contribution of the work is the suggested design for incorporating the unit checking as part of the elaboration (instantiation) process, which supports both implicit inference of unspecified dimensions and rational numbers of dimension exponents. To verify the design, a prototype implementation was constructed in the OpenModelica [17] environment.

The paper is structured as follows: Section 2 introduces fundamental terminology and describes design considerations affecting primarily the end user. Section 3 describes design issues from a library and tool perspective. Both these sections explore the design space in which a specific design can be created. Section 4 specifies a number of design choices made for a prototype implementation created in the OpenModelica environment. Section 5 discusses related work and section 6 concludes the paper.

¹In the remainder of the paper, the term *unit checking* will be used for *dimensional checking* as well. However, note that even if a system is dimensionally consistent, it might have conflicting units of measure.

2 End User Perspective

In this section, several aspects of unit checking will be discussed primary from an end user perspective. The section starts by refreshing fundamental terminology; followed by description of concepts such as type checking and polymorphism.

2.1 Units, Quantities, and Dimensions

Physical *quantities* are organized into different *dimensions*, such as length, time, and mass. The SI-system [7] defines seven *base quantities*, which can be combined to form new *derived quantities*.

For a particular quantity, there exist several different *units*, e.g., the quantity length can be used with both of the units meter and foot. To convert between different units within the same quantity dimension, *conversion factors* are defined. To convert from foot to meter a *scale factor* of 0.3048 is multiplied to the measured value. However, some unit conversions are more complex. For example, the formula $T_{Celsius} = (5/9) * (T_{Fahrenheit} - 32)$ for converting Fahrenheit to Celsius involves both a scale factor of 5/9 and an *offset* of value $-32 * (5/9)$.

The SI-system defines seven *base units* (m, kg, s, A, K, mol, cd) as well as *derived units*, which are accepted within the SI-system. These derived units have specific names and symbols and always have a corresponding normalized form expressed in base units. For example newton meter has the symbol N m, which has the expression $m^2 \text{ kg s}^{-2}$. For some derived quantities, the dimensional exponents are zero. Such a quantity is referred to as *dimensionless* or having dimension one. For example the derived quantity plane angle with derived unit radian is such a dimensionless quantity.

In Modelica, there is a syntax to define derived unit using base unit expressions. For example, the above expression of newton meter can be expressed as "m2.kg.s-2". From now on, this syntax will be used for describing unit expressions.

2.2 Static Unit Type Checking

When simulating Modelica models, the state of a dynamic model changes during the simulation, but the relation between the units of variables should not change dynamically².

²Using algorithms and functions, it is possible to define expressions that violates this principle. However, it would require the theory of dependent types to manage this property statically.

Hence, unit and dimensional checking can advantageously be performed statically at compile time. This process is typically accomplished by using a *static type checker*, which takes a Modelica model as input and returns one of three possible answers:

- *Consistent and complete.* The equations, connectors, hierarchy composed components, and the declared derived physical units match without exception. All variables have a specific unit assigned to it.
- *Consistent and incomplete.* The model is consistent (no conflicting constraints), but some variables have no units assigned to them.
- *Inconsistent.* One or several relations mismatch. For example, an equation $a = \text{der}(b) * 33 + c$ is inconsistent if a and c do not have the same units, or if the unit of b multiplied by "s" (time) is not equal to the unit of c.

A language and type checker can be designed to *infer* missing unit types, which can result in both a consistent and an inconsistent result.

Furthermore, from a user's point of view, it is important to *know* that the model is consistent, e.g., that the type checker can *guarantee* that unit errors do not exist. The property that a tool cannot find any inconsistencies in a model, does not imply that the model is consistent. In our proposal, this is a strong requirement for the design of the unit checker.

2.3 Detecting Errors, Isolating Faults

The previous described approach for unit checking enables *detection* of modeling errors, i.e., to give a sound judgement of the model's correctness regarding physical units and quantities. However, even if a tool can respond that a model is incorrect, it is very important for the user to know where in the model the fault is located. Hence, the tools' ability to *isolate faults* in a model is critical for making the unit checking process useable.

2.4 Polymorphism

A language where an object only can be of one type is said to have a *monomorphic* type system. This leads to a very restrictive language, with limited expressiveness. Modelica is a *polymorphic* language, where polymorphic behavior is primarily expressed using subtyping polymorphism.

Consider the following example of the block `Gain`, defined in the Modelica standard block library.

```
block Gain
  parameter Real k(unit="1") = 1;
public
  Interfaces.RealInput u;
  Interfaces.RealOutput y;
equation
  y = k*u;
end Gain;
```

Both input and output to and from the model are defined using `Real` types, i.e., no units are defined for this block. If a unit checker should be able to check instances of this block, unit types must be specified for its formal parameters. For example, both input and output can be defined to have unit type `Voltage`. However, this would result in a new block definition for every imaginable unit, which clearly is impractical.

A solution to this problem which is being implemented in this proposal is the use of *unit type variables*, and so called *parametric polymorphism* i.e., the block is declared to take a unit type variable 'p' as both input and output. Hence, the unit information is propagated from the input to the output³. This approach is similar to ordinary type variables used in for example Haskell [16] or Standard ML [11].

For general information about types and polymorphism see [3]. An accessible description on how types are related to Modelica can be found in [2].

3 Design from Library and Tool Perspective

This section presents requirements and a proposed design for unit checking from the perspective of implementers of libraries and tools.

3.1 Unit Type Declaration

There are two approaches of handling declaration of unit types, *implicit* unit type inference or *explicit* type declaration.

- Implicit type inference means that the user does not specify units for all variables and that the tool uses type inference to deduce the units of those variables.

³Note that parameter `k` needs to be explicitly defined to be dimensionless (`unit="1"`) in order to make a unit type inference algorithm to work. If it was left as unspecified, the gain could generate any possible unit, regardless of its input.

- Explicit type declaration means that the user specifies units for variables, and thus removes the need of deducing units.

For instance, consider the following example:

```
model A
  Real(unit="m") x1,y1,d1,d2;
  Real x2,y2;
equation
  d1 = sqrt(x1^2+y1^2);
  d2 = sqrt(x2^2+y2^2);
end A;
```

The example calculates the distances of two points to the origin $(0, 0)$. The first point (x_1, y_1) uses explicit unit type declaration, giving `x1,y1` and `d1` the unit "m", and the second point (x_2, y_2) uses implicit type inference, where units are not specified. In the second case the units can be deduced from the unit of the distance variable `d2`, i.e., the unit type of `x2` and `y2` are *inferred* from the unit type of `d2`.

A problem is how to distinguish between dimensionless units and implicit type inference. Consider the following declaration:

```
Real x;
```

Is `x` dimensionless or should the type be inferred (i.e., has *any* dimension)? The most probable interpretation is that it should be inferred. There are several alternatives of how to declare a dimensionless unit. One solution is to use

```
Real x(unit = "1");
```

It is important to differentiate between any dimension and dimensionless, because the distinction can give better information for the unit checker to perform its task.

To be able to handle parametric polymorphism it must be possible to declare *unit type variables*. A unit type variable can hold any unit type and thus provides flexibility of e.g., writing functions. For instance, consider the following example:

```
function myDer
  input Real x(unit="'p");
  output Real y(unit="'p.s-1");
algorithm
  y:= der(x);
end myDer;
```

The example is a wrapper around the `der` operator. The unit of the input argument uses a unit type variable " 'p " which is used to express the unit of the result from the function. Here the character ' is part of the type variable identifier and indicates that this is a type

variable and not a normal variable. Using a type variable makes it possible to use the `myDer()` function for any type of unit, and still being able to express the relation between the unit types of the input and output argument.

3.2 Unit Conversion

For many situations it is necessary to convert expressions from one unit to another. A unit conversion does not change the dimension of an expression, only its value. For instance:

```
SI.Length d1 = 25.4;
Real d2 =
  unitConvert(d1, "mm");
```

For this case 25.4 is interpreted as meter (defined in `SI.Length`). The proposed built in function `unitConvert(var, unit)` converts the value to 25400 and assigns it to `d2`. Moreover, `d2` is now assumed to have unit "mm". Note that it is not possible to just scale this using an ordinary multiplication, since the user must tell the type checker that the unit has been changed.

In conclusion, unit conversion is a fundamental requirement to be able to work conveniently with units.

3.3 Representation of Units

The unit checking mechanism requires the tool to be able to distinguish between different (base) units. This is typically solved (e.g., in [14, 15]) by having a vector of seven base units, as described by the SI standard [7]. For instance, energy can in the SI units be described using "J" (Joule) or "N.m" (Newton meter) corresponding to the base unit " $\text{m}^2 \cdot \text{kg} \cdot \text{s}^{-2}$ ". Currently, a Modelica tool would need to know that "J" or "N.m" correspond to the base unit " $\text{m}^2 \cdot \text{kg} \cdot \text{s}^{-2}$ " and how to construct the appropriate vector for such a unit.

To be able to handle functions like calculating the square root of a value (the `sqrt` function), the coefficients of the dimension vector must be able to handle more than integer numbers. By using rational numbers instead it is possible to express e.g., the square root with exponent (1/2). Note that it is not possible to use floating point precision as coefficients, since that would lead to roundoff errors.

A problem related to the representation of units is how to present a unit to the user. Often a user has no idea what the unit " $\text{m}^2 \cdot \text{kg} \cdot \text{s}^{-2}$ " means. Instead, the user expects the derived unit to be output, i.e., "N.m". The problem of unparsing (pretty printing) the internal unit

representation to a string must be considered. Often, the choice of derived units to use is not obvious, and heuristics must be used to achieve what a user might expect as output. Such heuristic is not trivial to do and it might even be different depending on the context (application area) of the user model.

3.4 Defining Units in the Modelica Language

To be able to handle other units than those described by the SI standard, a more elaborate design than using seven base units must be introduced. For instance, a financial institute involved in modeling and simulating the stock market might be interested in using the quantity "money". Also, they would like to be able to add scaling factors between different units of money (\$, €, SEK, etc.). Thus, an important design requirement for the unit checking framework is that the number of base units is not known a priori, i.e., end users must be able to add whatever units they want. Also, the scale and offset information must be available for the unit checking module. Finally, it must also be possible to describe the relation between base units and derived units.

Currently, Modelica does not have support for adding scaling (and offset) for units, neither can one add ones own "base units". Today, Modelica has some knowledge about the SI units, e.g., a Modelica tool with unit checking capabilities knows that `unit="m"` refers to the base unit meter and `unit="F"` refers to the non-base unit farad (expressed as " $\text{m}^{-2} \cdot \text{kg}^{-1} \cdot \text{s}^4 \cdot \text{A}^2$ " in base units) and not to Fahrenheit. But, if users should be able to add their own base units, the language should instead be extended so that base units can be described in Modelica. The SI-units package would then first declare the SI base units, and then derive units based on these base-units.

Moreover, information for converting between units is not covered by current Modelica. To be able to convert between different units, scaling and offset information must be introduced. For instance, consider converting between Fahrenheit and Kelvin. This can be achieved using a scaling factor and an offset as illustrated by the conversion function in the standard library:

```
function from_degF
  input NonSIunits.Temperature_degF
    fahrenheit;
  output Temperature kelvin;
algorithm
  kelvin := ((fahrenheit - 32)*5)/9 -
    Modelica.Constants.T_zero;
end from_degF;
```

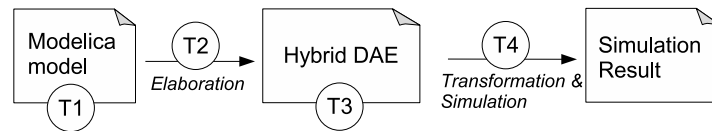


Figure 1: Possible unit checking-times (T1,T2,T3,T4) during the Modelica compilation and simulation process.

If the scale and offset information instead is added to the unit types (e.g., as attributes to the built-in Real class), such conversion functions would not be required. Instead the tool could perform the conversion using the built-in `unitConvert()` function, rendering convert functions in the standard library redundant.

3.5 Time of Checking

There are several different points in time during the translation process where the unit checking mechanism could be introduced, see Figure 1.

- T1 - At the model level.
- T2 - During elaboration.
- T3 - At the hybrid DAE (flat Modelica) level.
- T4 - During runtime/simulation.

Some checks can be made at the model level (T1), performing checks for each individual sub-model. Local equations in the model can be checked this way, but not equations generated from connecting components together, or components where types must be deduced from the surrounding environment (e.g., connections or modifiers). Another approach is to combine the unit checking phase with the elaboration (flattening) process (T2).

Checking on the flat model (T3) is of course feasible, leading to a large check of the overall system. The advantage of this approach is its simplicity; a translation of the model into equations for the unit checking module is performed only once. The disadvantage is that it is much harder to isolate the fault, since only the flat set of equations is available. Also, this approach will not make use of already checked parts, e.g., checking the model equations of an electrical resistor will be done not only once but for as many times as the resistor model is used as a component. The gPROMS unit checking tool [14, 15] uses this approach. Finally, some analysis cannot be performed statically and must then be performed during runtime, i.e., during the simulation (T4).

4 Prototype Implementation

A prototype implementation based on the design requirements presented above is under development in the OpenModelica[17] and MathModelica[10] compilers. The compiler does a static (during compilation) check of dimensions and units of measure.

4.1 Design

The design includes the following aspects:

- Rational numbers as exponents on dimensions.
- Unit type variables in declarations.
- Literal constants are treated differently depending on context (dimensionless in multiplication/-division and unknown in addition/subtraction).
- Type inference of dimensions.
- User defined base and derived units.
- Checking is performed during elaboration / flattening to enable better fault isolation.

The design is split into separate parts, see Figure 2.

One part is integrated with the elaboration (flattening) process in the OpenModelica compiler. It will create an equation systems to be solved by the Unit Checker (the second part) for model components according to the same principles as components are instantiated in Modelica (i.e., a recursive process). This is done by first adding units to a unit store by calling the `addStore` function in the `UnitASTBuilder` module. Next, local equations are traversed to build unit terms, with the `buildTerms` function. Both the unit store and unit terms are defined in the `UnitAbsyn` module. Finally, the `check` function in the `UnitChecker` module is called to perform the dimension analysis. The result from the checking of each component contains two pieces of information. First, for each component it will receive an answer whether a component is Ok (consistent and complete), inconsistent (incompatible types) or consistent and incomplete (not enough information available). Secondly, it will calculate the resulting unit type variables of a component which can then be used

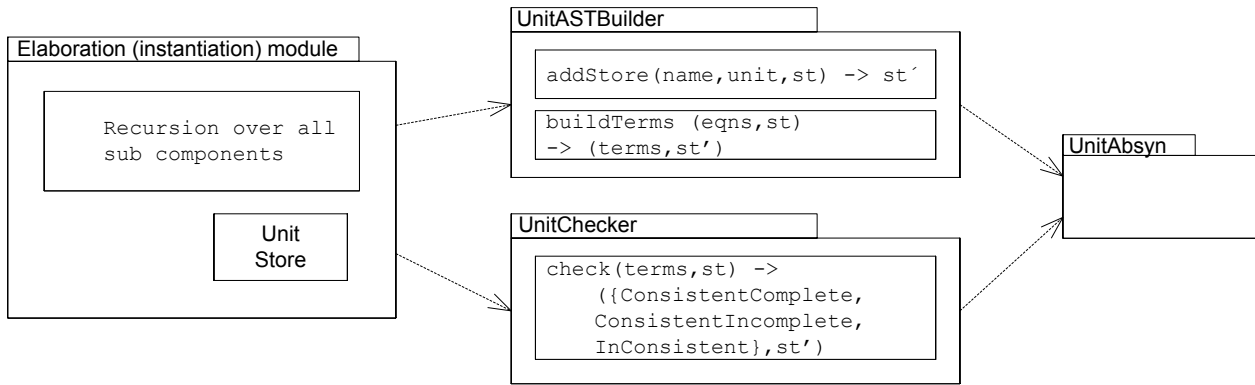


Figure 2: Outline of the main modules of the unit type checking engine of the prototype implementation. Arrows describe dependencies between modules.

when checking the complete model. This will give the following steps of the unit checking function.

1. Check components in the class.
2. Build a new equation system from the type variables from each component together with local equations and connections.
3. Call the unit checker for the model itself.

Note that checking the components of a class means a recursion over the three steps for the class of the component.

The equation systems for the unit checker are created from two data structures, a unit store that holds units of variables, and unit terms that describe constraints between different variables. The following sections show how these are built.

4.1.1 Storing Units

Each variable in a model has a corresponding unit. A unit can be

- A specified unit, e.g., "m/s".
- A unit type parameter e.g., "'p", with an optional exponent, e.g., "'p^2".
- A combination of specified unit and type parameter, e.g., "'p/s".
- unspecified unit e.g., the unit of a declaration "Real x;".

The unit store is a data structure that holds the units of variables. It gives a mapping from a variable name

to its corresponding unit. During the instantiation and unit checking process the unit store is updated with new units. The following model shows how the unit store is used:

```

model SimpleOde
  Real x;
  Velocity v;
equation
  der(x)=2*v + 1.0;
end SimpleOde;

```

First the unit store is built by adding the units of the variables x and v . Since x is declared as a Real it gets an unspecified unit, and v gets the unit "m/s".

After the unit checking module has been executed on this class, it will update the unit store with the unit for x with "m", because this was inferred by the UnitCheck module. This information can then be used higher up in the instance tree to check units of other components.

4.1.2 Building Unit Terms

The second data structure required for building unit constraint equations is the Unit Term which describes relations between variables. This structure is similar to the data structure for equations, containing nodes for e.g., addition, multiplication, etc. It is sufficient to only have four types of relations between units: multiplication of terms, division of terms, addition of terms, and equality between terms. Since an addition of two variables and a subtraction of two variables both imply the same rules for the units, both of these can be expressed using the same unit term. The leaf nodes of terms are references to units in the unit store.

Let us again consider the example SimpleOde above. We use ADD and MUL for addition and multiplica-

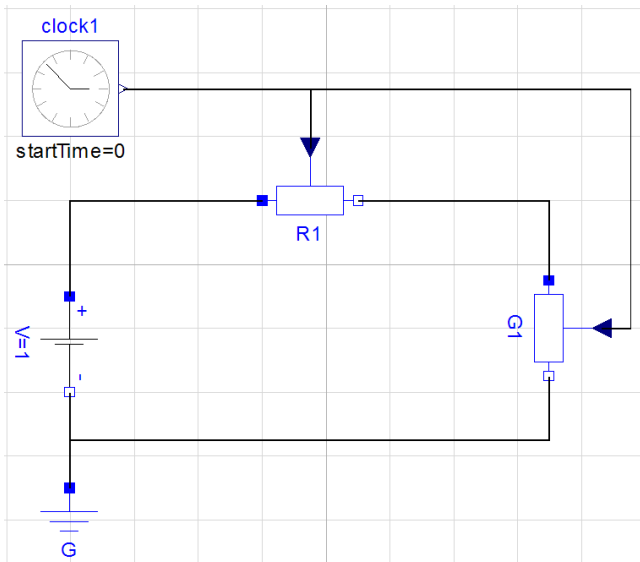


Figure 3: An inconsistent circuit that should fail during dimensional checking.

tion in our data structure and EQN for equality between terms. For the leaf nodes, with references to the unit store, are described with LOC. The example above corresponds to the following terms (somewhat simplified):

```

EQU(
  LOC("der(x)"),
  ADD(
    MUL(LOC("V"), LOC("2")),
    LOC("1.0"))
)

```

From the unit store and the unit terms, constraint equations are built. A multiplication of unit terms means that the unit vector is added, and an addition of unit terms means that the units must be equal.

4.1.3 Built-in Functions and Operators

The built-in functions and operators are extended with units containing unit type parameters. That gives us a uniform way of dealing with functions, regardless if the function is a built-in function, a built-in operator, or a user-defined function. For instance, the der operator is internally described as

```

function der
  input Real x(unit = "p");
  output Real y(unit = "p/s");
  external "builtin";
end der;

```

That is, applying the derivative operator to an expression will change its unit by multiplication with "s-1".

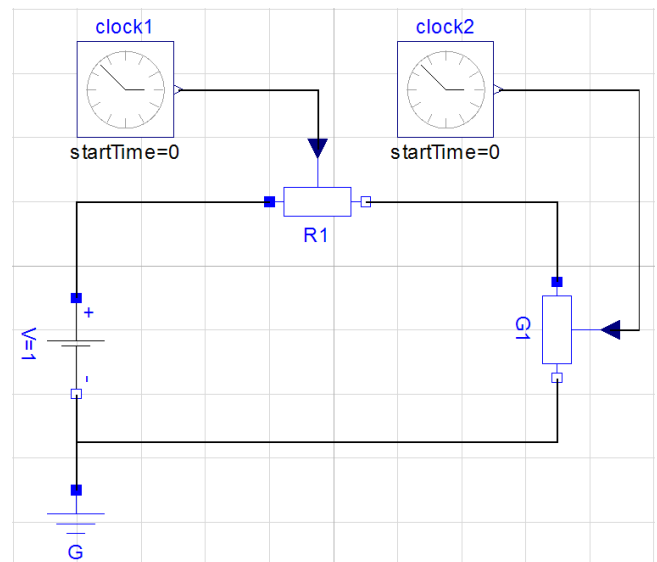


Figure 4: A dimensionally correct circuit.

4.2 Example

Let us consider an example using components from the Modelica Standard Library to illustrate the different aspects of unit checking. Figure 3 shows an example where unit checking will return an error because of inconsistent units⁴. A VariableResistor and a VariableConductor is fed from the same signal source, taken from the Blocks library. All sources in the Blocks library have unspecified units, such that they can be used in any context. The unit checker will find that the unit of the output of the clock generator should be both "Ohm" (Resistance) and "S" (Conductance), i.e., an inconsistency is reported. This inconsistency is detected first when the local equations of the Circuit model is unit type checked. The unit store then contains an unspecified unit for the clock generator (clock1.y) and specified units for the inputs on the resistor R1 (R1.R) and the conductor G1 (G1.G).

To resolve the inconsistency of the circuit the user has to use two separate clock generators, see Figure 4. The unit of clock1.y will become "Ohm" and clock2.y will become "S", resulting in a consistent system.

When using math blocks (Gain, Add, TransferFunction, etc) in models it becomes evident that polymorphism is required. For instance, lets add a gain to

⁴The circuit is inconsistent since the VariableResistor and VariableConductor have declared their inputs to Resistance and Conductance respectively. If they were declared as dimensionless the circuit would have been consistent, thus also making it a library design issue. Also, it could be possible to have different unit checking semantics depending on the causality of equations, which would allow this kind of connections.

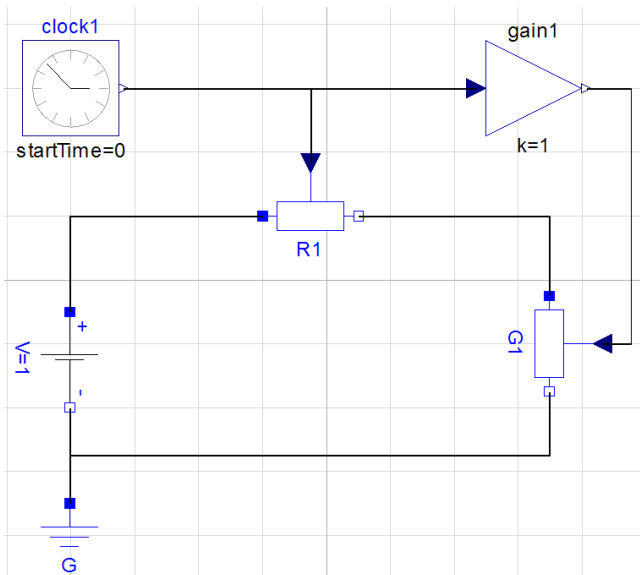


Figure 5: An inconsistent model with a polymorphic block.

our inconsistent model, see Figure 5. The gain block should be possible to use for any unit, i.e., it should be a polymorphic block. If that would not be possible, the user would have to write a new block model for each particular use, in this case for amplifying a Conductance signal. In our implementation, the unit checker will treat the Gain block as having a polymorphic unit and assign a unit type parameter to it. The result of checking the gain block is a unit type parameter that propagates the unit of the input to the unit of the output. Hence, when the circuit model is checked, the unit from the VariableConductor is propagated to the unit of `clock1.y`, leading to an inconsistent system of equations. Typically, for larger block models, this propagation can be performed over many subsystems of components. This implementation will however lead to a detection of the inconsistency at the lowest level possible, making it easier for the user to correct the inconsistency.

5 Related Work

Unit checking has been introduced in several Modelica tools over the last couple of years, for instance, Dymola[4] from Dynasim and Simulation X[8] from ITI GmbH. Dymola version 6.1 has a unit checking mechanism, as well as support for deduction of units. However, unit parametric polymorphism is yet not supported.

Simulation X has a conversion extension to Modelica for giving units to literals. For instance, the expression

`a + 2.5 'mm'` will translate the literal `2.5` into SI base unit meter by multiplying it with $10e-3$.

Both of these tools will (or soon will) support entering other units than default units for e.g., parameter values, i.e., making it possible to enter `2.5 mm` as a parameter value. The `displayUnit` attribute of Modelica standard is available for this purpose.

Unit checking and checking of dimensional inconsistency has been extensively explored in the programming language research community and is far from a new research area. Many library-based approaches exist for imperative programming languages, such as a package approach for Ada [6] and a template approach in C++ [18]. An approach for dimensional inference is presented in [19], where gaussian elimination is used for solving the resulting equation system. The work shows how dimensions with rational exponents can be added to the simply typed lambda calculus.

In Kennedy's thesis [9], an extension of a core calculus of ML with support for type inference over dimension types is given. Lately, dimension and unit checking have also been addressed in a nominally typed object-oriented language [1].

Besides the work on gPROMS [14, 15], few attempts have been made to incorporate dimensional and / or unit checking in equation-based object-oriented languages, such as Modelica. In addition, even though Modelica today supports syntax for stating units of variables, no sound solution exists that guarantees the absence of unit errors.

6 Conclusions

This paper has presented a design for dimensional analysis and unit checking of Modelica models. Requirements from an end user and tool perspective have lead to a design which has been implemented as a prototype on top of the OpenModelica and MathModelica compilers. MathModelica has also been used for building the models presented in this paper, and a future release of MathModelica will contain unit checking based on the design in this paper. The design introduces unit type variables enabling polymorphism of unit types in Modelica, which increase the safety and flexibility of the dimensional analysis. We have also chosen to represent exponents as rational numbers which enables dimensional checking of e.g., the `sqrt` function. The design of the dimensional analysis also allows the possibility of adding additional base units, on top of the seven base units of the SI system. This enables modeling of e.g., financial systems using

base unit money, and other application areas.

The prototype implementation has been described and illustrated with several examples from the standard library. The analysis results in either a consistent and complete system, a consistent but incomplete system (which means that not sufficient unit information is available to fully determine units) or an inconsistent system (indicating where the inconsistency is located). By using the prototype we have detected some minor problems with the standard library. For instance, the Gain component in the Blocks Math library currently has unspecified units on its gain parameter. In order to fully check the dimensions of models using this component, the gain parameter should be dimensionless. This paper has also discussed unit conversion, even though this has not yet been implemented. Nonetheless, some ideas presented here could be a useful starting point for the Modelica Design Group's activities regarding this topic.

Acknowledgments

This research was funded by CUGS (National Graduate School in Computer Science), MathCore Engineering, by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project, the Swedish Research Council (VR), and Forska&Väx program under the project Lättanvänt Generellt Simuleringsverktyg för Industriell Produktutveckling.

References

- [1] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Jr. Guy L. Steele. Object-Oriented Units of Measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, Vancouver, BC, Canada, 2004. ACM Press.
- [2] David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, pages 303–315, Vienna, Austria, 2006.
- [3] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.
- [4] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). <http://www.dynasim.se/> [Last accessed: Jan 22, 2008].
- [5] Narain Gehani. Ada's derived types and units of measure. *Software Practice and Experience*, 15(6):555–569, 1985.
- [6] Paul N. Hilfinger. An ADA Package for Dimensional Analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, 1988.
- [7] Bureau international des poids et mesures (BIPM). *Le Système international d'unités, The International System of Units*. Organisation intergouvernementale de la Convention du Mètre, 8th edition.
- [8] ITI. SimulationX. <http://www.iti.de/> [Last accessed: November 8, 2007].
- [9] Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catharine's College, University of Cambridge, UK, UK, 1996.
- [10] MathCore. MathModelica System Designer: Model based design of multi-engineering systems. <http://www.mathcore.com/products/mathmodelica/> [Last accessed: Jan 23, 2008].
- [11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
- [12] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0*, 2007. Available from: <http://www.modelica.org>.
- [13] Gordon S. Novak. Conversion of Units of Measurement. *IEEE Transactions on Software Engineering*, 21(8):651–661, 1995.
- [14] Daniel Persson. Dimensional Analysis and Inference for gPROMS. Master's thesis, Department of Computer Science and Engineering, Mälardalen University, Sweden, 2003.
- [15] Mikael Sandberg, Daniel Persson, and Björn Lisper. Automatic Dimensional Consistency Checking for Simulation Specifications. In *SIMS 2003*, September 2003.

- [16] Simon Peyton Jones. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
- [17] The OpenModelica Project. <http://www.ida.liu.se/~pelab/modelica/OpenModelica.html> [Last accessed: January 22, 2008].
- [18] Zerkis D. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(9):135–139, 1994.
- [19] Mitchell Wand and Patrick O’Keefe. Automatic Dimensional Inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic - Essays in Honor of Alan Robinson*. The MIT Press, 1991.

Unit Checking and Quantity Conservation

Sven Erik Mattsson Hilding Elmqvist
 Dynasim AB
 Ideon Science Park, SE 223 70 Lund, Sweden
 SvenErik.Mattsson@3ds.com Hilding.Elmqvist@3ds.com

Abstract

What can be done to guaranty correctness of a model? The paper discusses two approaches to automatic checking. First, Dymola's support of units, unit checking and unit deduction is described. It has already proven useful and has helped improving the quality of the Modelica Standard Library. The display unit concept allows users to enter parameters and plot variables in different units. The inputs, outputs and parameters of general blocks defining sources and mathematical operations have of course no units specified. Dymola infers their units in order to improve the variable browsers for entering parameter values and plotting variables during simulation. Second, the possibilities of checking quantity conservation automatically are discussed. It is in an open area with a large potential to check that models fulfill the very basic laws of physics including energy conservation, Newton's third law "action equals reaction", etc. To really support automatic checking of quantity conservation it is necessary to include more information in the models. Fortunately, it seems as if most of this can be done in the basic components such as inertia, body, volume, capacitor etc which actually store some quantities and in dissipative elements as for example resistors or friction elements.

1 Introduction

Modelica (Modelica, 2007) is a powerful modeling language. It allows you to quickly build complex models by putting together model components from free public and commercial libraries. The openness of Modelica makes it easy to modify an existing component. All this opens for errors. How can we guide users and provide automatic checking? How can we guarantee quality of provided library components?

Modelica is a strongly typed language implying that classical computer scientific methods can be used.

The Modelica 3.0 definition has taken this further and introduced the concepts of plug-in compatibility and balanced models. This paper will discuss two other orthogonal approaches:

1. unit checking of expressions and equations
2. checking of quantity conservation.

In Sections 2-4, Dymola's support of units, unit checking and unit deduction is discussed. In Section 5 the possibilities of checking quantity conservation automatically are discussed.

2 Support of Units in Dymola

Physical modeling deals with physical quantities such as length, mass, force, current. The value of a quantity is generally expressed as the product of a number and a unit. Modelica (2007) supports this approach. A real variable have a quantity attribute and a unit attribute, for example

```
type Mass = Real(quantity="Mass",
                 final unit="kg");
```

The package Modelica.SIunits provides a large set of predefined quantities and it is recommended to use them whenever possible.

2.1 SI units

The Modelica specification states "A basic support of units in Modelica should know the basic and derived units of the SI system." Dymola fulfils this requirement.

A good reference on SI units is what commonly is called the SI brochure published by Bureau International des Poids et Mesures [BIPM, 2006]. The NIST Reference on Constants, Units, and Uncertainty [NIST, 2000] gives a good overview; see also [Taylor, 1995]. ISO does not specify a formal syntax for unit expressions but there are strict recommendations. The Modelica language specification includes a formal specification based on these recommendations.

Dymola supports all the 20 SI prefixes to form decimal multiples and submultiples of SI units.

Factor	Name	Symbol	Factor	Name	Symbol
10 ¹	deca	da	10 ⁻¹	deci	d
10 ²	hecto	h	10 ⁻²	centi	c
10 ³	kilo	k	10 ⁻³	milli	m
10 ⁶	mega	M	10 ⁻⁶	micro	μ
10 ⁹	giga	G	10 ⁻⁹	nano	n
10 ¹²	tera	T	10 ⁻¹²	pico	p
10 ¹⁵	peta	P	10 ⁻¹⁵	femto	f
10 ¹⁸	exa	E	10 ⁻¹⁸	atto	a
10 ²¹	zetta	Z	10 ⁻²¹	zepto	z
10 ²⁴	yotta	Y	10 ⁻²⁴	yocto	y

Dymola knows all the seven SI base units

Name	Symbol
metre	m
kilogram	kg
second	s
ampere	A
kelvin	K
mole	mol
candela	cd

as well as the 22 SI derived units that have been given special names and symbols

Name	Symbol (in Modelica)	Definition
radian	rad	1
steradian	sr	1
hertz	Hz	1/s
newton	N	kg.m/s ²
pascal	Pa	N/m ²
joule	J	N.m
watt	W	J/s
coloumb	C	A.s
volt	V	W/A
farad	F	C/V
ohm	Ohm	V/A
siemens	S	A/V
weber	Wb	V.s
tesla	T	Wb/m ²

henry	H	Wb/A
degree Celcius	degC	K
lumen	lm	cd.sr
lux	lx	lm/m ²
becquerel	Bq	1/s
gray	Gy	J/kg
sievert	Sv	J/kg
katal	kat	mol/s

There are also units that are not part of the International System of Units, that is, they are outside the SI, but they are accepted for use with the SI. Dymola knows the following of them:

Name	Symbol	Expressed in SI units
minute	min	60 s
hour	h	60 min
day	d	24 h
degree	deg	(π/180) rad
litre	l	dm ³
decibel	dB	1
electronvolt	eV	0.160218 aJ
bar	bar	0.1 MPa
phon	phon	1
sones	sones	1

In power systems the unit for apparent power is “V.A”. Dymola knows var = V.A which has been adopted by the International Electrotechnical Commission, IEC, as the coherent SI unit volt ampere for reactive power, see IEC [2007].

The rotational frequency n of a rotating body is defined to be the number of revolutions it makes in a time interval divided by that time interval. The SI unit of this quantity is thus the reciprocal second, s⁻¹. However, the designations "revolutions per second" (r/s) and "revolutions per minute" (r/min) are widely used as units for rotational frequency in specifications on rotating machinery. Although use of rpm as an abbreviation is common, its use as a symbol is discouraged. Dymola knows $r = 2\pi$ rad. It can be used for example as “r/s” or “r/min”.

Dymola also knows the temperature units degF (degree Fahrenheit) and degRk (degree Rankin).

2.2 Other units

Dymola recognizes the users' needs to enter parameters and plot variables in different units. Modelica

defines `displayUnit` for that purpose. Dymola supports `displayUnit` when plotting variables and when entering values in parameter dialogs.

A user can define units for display and its meaning in terms of the SI unit. For example, the display unit “min” is defined in the following way in terms of the SI unit “s” as

```
defineUnitConversion("s", "min", 1/60);
```

There is a fourth optional argument to specify offset. For example, conversion from Kelvin to degrees Fahrenheit can be specified as

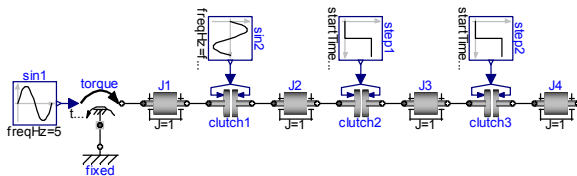
```
defineUnitConversion("K", "degF",
  9.0/5.0, 32-(9.0/5.0)*273.15);
```

However, if the quantity represents a temperature difference the offset shall not be included. Dymola supports an annotation `__Dymola_absoluteValue` to control this. In Modelica.SIunits the quantity temperature difference is specified as

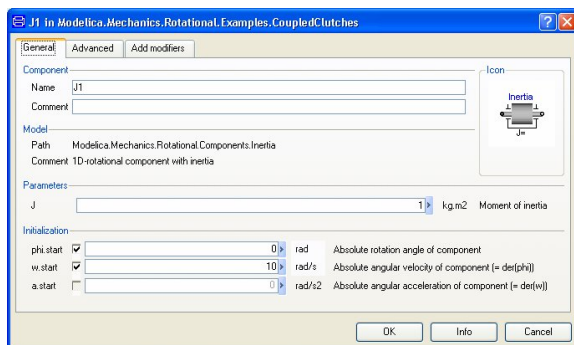
```
type TemperatureDifference = Real (
  final quantity=
    "ThermodynamicTemperature",
  final unit="K")
annotation
  (__Dymola_absoluteValue=false);
```

These definitions are conveniently stored in script (mos) file that is executed at the start of Dymola. By default Dymola has a file `displayUnit.mos` including display units of general interest.

As an example, consider the model `CoupledClutches` in Modelica Standard Library 3.0.



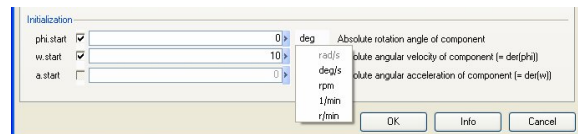
Pop the parameter dialog for the rotating body, J1.



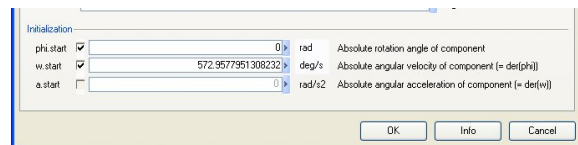
The start values for the angle, ϕ , and the angular velocity, w , can be entered in SI units. The Modelica code for J1 is

```
Modelica.Mechanics.Rotational.Inertia
J1 (J=1,
  phi(fixed=true, start=0),
  w(start=10, fixed=true))
```

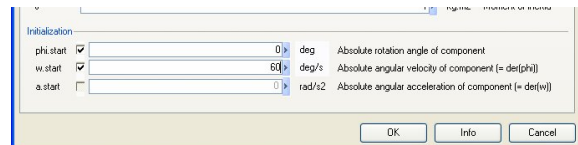
We can enter the start velocity in “deg/s”. Click on the unit to pop a menu and select unit



Which alternatives that are available depends on which `defineUnitConversion` calls that actually have been invoked. It can be customized by any user by editing the file `displayUnit.mos`. If a user wants to see any length only in mm or inch, the user can restrict the display unit to that.



The value is now displayed as 572.96 deg/s. It is easy to enter a new value, say 60 deg/s.

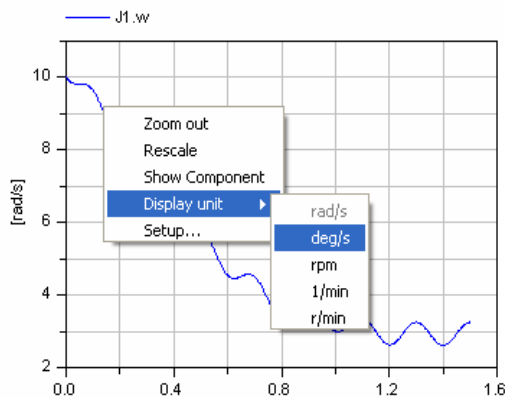


The Modelica code for J1 becomes

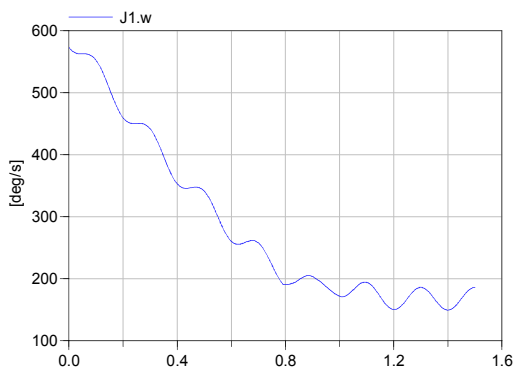
```
Modelica.Mechanics.Rotational.Inertia
J1 (J=1,
  phi(fixed=true, start=0),
  w(start=1.047197551196598,
    fixed=true, displayUnit="deg/s"))
```

Note, that the attribute `displayUnit` is modified according to our choice. However, the parameter value being 60 deg/s is stored in SI units, “rad/s”. Thus portability is preserved and it is still a tool issue to support the `displayUnit` in the dialogs.

Let us simulate the original example and plot J1.w. Put the cursor on the curve and pop the context menu.



Selecting “deg/s” as unit for plotting gives the plot.



3 Unit Checking

Equations add terms. Naturally these must be of the same physical quantity. This is exploited in the classical physical dimension check of equations which many of you have done by paper and pen in school. Dymola (Dynasim, 2007) has automated this check.

The number of physical quantities we can think of is large. Fortunately, they are related and all physical quantities can be expressed as product of powers of a small set of base quantities. The International System of Units, the SI system, defines such a set including seven physical quantities: length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity, see BIPM (2006). The SI base units define a unit for each of these seven quantities. The units for other quantities are derived. For example, the unit for area is m^2 because the physical quantity $area = length * length$ and the unit for length is m (meter). Thus there is a mapping from quantity to unit in terms of the seven SI base units. Dymola exploits this for unit checking.

Dymola’s checking of units is active when checking a package, function or model as well as when translating a model for simulation. It includes checking of

unit strings and unit compatibility of equations. It can be seen as a part of the type checking. It includes the checking of actual function input arguments and output arguments against their formal declarations.

Currently Dymola makes a relaxed checking. It means that an empty unit string, "", is interpreted as unknown unit. Also number literals are interpreted to have unknown unit. The unknown unit is propagated according to simple rules

unknown unit * "unit1" -> unknown unit
 unknown unit + "unit1" -> "unit1"

There is one important exception. Let e be a scalar real expression. Consider the inverse of e given as $1/e$. The number 1 (one) in the numerator does not relax the checking. If e has a well-defined unit then also $1/e$ has a well-defined unit.

The unit checking is applied to the original equations. This has implications for vector, matrix and array equations. For an array where all elements have the same unit, the check works as if it was a scalar. Arrays and array expressions where the elements have different units are allowed. However, the check is then relaxed and the array is viewed to have an unknown unit that is compatible with all units. Checking the unit consistency between two records is done recursively for each component.

Currently, the unit checking does not issue error messages but it generates only warnings. The unit checking can be disabled.

As a simple example consider the modeling of motion where there is a mistake

```
parameter Modelica.SIunits.Mass m=1;
Modelica.SIunits.Velocity v;
Modelica.SIunits.Force f;
equation
m*v = f; //Should read m*der(v) = f;
```

When checking or translating it, Dymola outputs

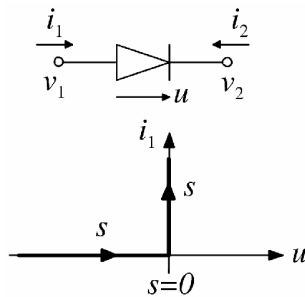
```
Warning: Incompatible units in
m*v = f;
The part
m*v
has unit N.s
The part
f
has unit N
```

Dymola’s unit checking has already been proven useful. Several errors in the Modelica Standard Library were found. A user reported that he several years ago had rewritten a model in Modelica, but he did not get the same simulation result. He had really

tried to find the reason, without success. Dymola's unit checking pointed out an inconsistency and he had found the error.

The basic laws for conservation of mass, momentum, electrical charge, energy are expressed as balance equations between physical quantities. Also constitutive equations such as Ohm's law are readily expressed as equations for physical quantities. It means that the unit checking should not make the modeling more complicated in most cases.

However, the parameterized curve descriptions used to model idealized characteristics of for example diodes or Coulomb friction needs more attentions. Consider the modeling of an ideal diode having the characteristics shown in the figure.



The parameterized curve description is

```
off = s < 0;
v = if off then s else 0;
i = if off then 0 else s;
```

The curve parameter is just a real variable that is either representing a voltage or a current. To make the equations unit consistent, the equations can for example be rewritten as

```
v = unitVoltage*(if off then s else 0);
i = unitCurrent*(if off then 0 else s);
```

The s parameter and the unit constants are declared protected as

```
protected
Real s(final unit="1");
constant Modelica.SIunits.Voltage
unitVoltage= 1
annotation(HideResult=true);
constant Modelica.SIunits.Current
unitCurrent= 1
annotation(HideResult=true);
```

The HideResult annotation has the effect that the unit constants are not included in the simulation result. Basically these constant are only active during unit checking and then eliminated in the equations.

In summary, don't just declare real variables, but declare physical quantities. Use the predefined quantities available in Modelica.SIunits whenever possible. The SI units were invented to allow equations to be written in a clean way without conversion factors. This simplicity is a very good reason for using the SI units in physical modeling. Thus, it is recommended that unscaled SI units are used when specifying the unit attribute of a real variable. To be clear, this also means that prefixes shall not be used. For example "m", "kg", "V", "N.m" and "W" are good, but not "cm", "g", "kV", "MW" or "bar". The displayUnit concept provides convenient entering of parameter values and displaying and plotting of results in other units.

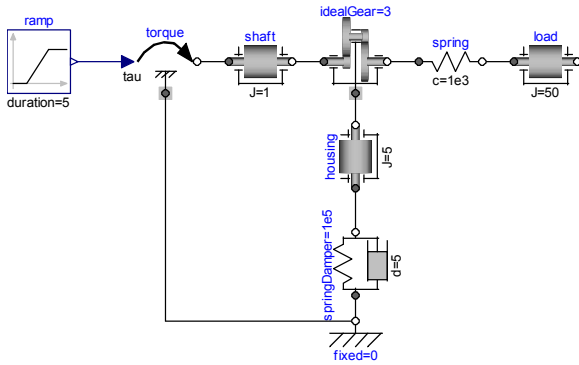
4 Unit Deduction

The Modelica.Blocks library includes general blocks to define sources and mathematical operations. Their inputs and output have of course no units specified. For user convenience, Dymola has introduced automatic deduction of units. Here is a short description Consider the expression, $e1 + e2$, where Dymola has found that the expression $e1$ has a well-defined unit $u1$, but the unit of the expression $e2$ is unknown. We can then deduce as described in the introduction that the unit of the sum $e1 + e2$ is $u1$. Moreover, for unit consistency reasons the unit of $e2$ must also be $u1$. If now $e2$ is a simple variable reference, v , we can deduce that v must have the unit $u1$. For more complex expressions Dymola makes a downwards recursion to see if it is possible to deduce units of variables with unknown units.

The SignalType definition in the Modelica Standard Library 2.0 allowed the user to specify the units manually by declaring the type of the inputs, the outputs and the parameters of the block. The SignalType is removed in the Modelica Standard Library 3.0 and the units are deduced automatically.

The deduction of units may reveal unit inconsistencies. In such a case it may be useful to enable the logging and inspect the log. It is also useful to check the log when developing a model component, because if a real variable gets its unit deduced that may indicate that the variables shall be declared using any of the quantities defined by Modelica.SIunits.

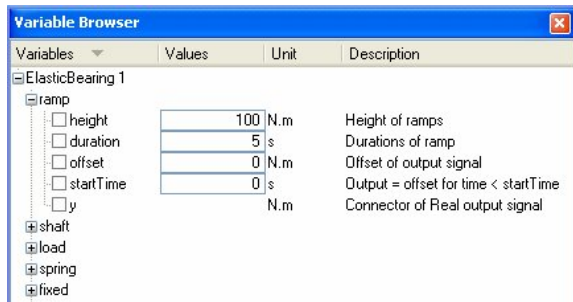
As an example consider the model ElasticBearing in Modelica.Mechanics.Rotational.Examples.



The component ramp is of the class Modelica.Blocks.Sources.Ramp. The parameters

```
parameter Real height=1
    "Height of ramps";
parameter Real offset=0
    "Offset of output signal";
```

have no units specified. Similarly the unit of the output y is not specified. At translation Dymola deduces their units and displays them in the variable browser



The deduced SI unit radian is treated in a special way by Dymola. Consider Euler's equation for a one dimensional rotating body

$$J \cdot a = \text{flange_a.tau} + \text{flange_b.tau};$$

where the inertia, J, has unit kg.m², the rotational acceleration, a, has unit rad/s² and the torques flange_a.tau and flange_b.tau have the unit N.m. It means that the left hand side of the equation has the unit, rad.kg.m²/s² and the right hand side has the unit, N.m = kg.m²/s². It means that the units are equal besides the left side has a factor "rad". This is fine from the formal point of view because the derived unit radian is formally expressed as m/m, see Table 3 in [BIPM, 2006], which also states that the radian is "a special name for the number one that may be used to convey information about the quantity concerned."

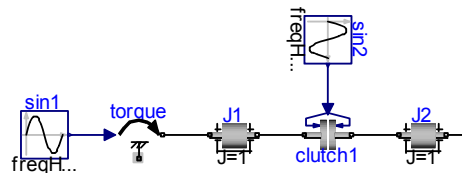
However, in order to support the use of radians when deducing units, Dymola treats the radians as if it was a SI base unit during the analysis. The consistency checking is of course relaxed for radians. The resulting unit will include the minimum power of radians.

5 Quantity conservation

The design of Modelica.Mechanics.Rotational and the discussion on the modeling of mounting have clearly indicated the need for more automatic testing of models. The failure to model the mounting of a drive-train element is an example where the user fails to account for important interactions between components or between a component and its environment. The failure to model the mounting of a component gives a simulation result where momentum is not preserved. Such a model violates Newton's third law "action equals reaction".

Balance equations and conservation of physical quantities such as mass, momentum, energy and electrical charge are basic in physical modeling.

A flow variable of a connector represents the flow of a conserved quantity into a component. Thus it is straightforward to calculate the net amount of a conserved quantity flowing into a component. A prototype implementation has been made in Dymola. As an example consider the model



The flow variables of the connectors have their attributes quantity="Torque". Dymola introduces for each component a variable named sum_Torque and an equation such as

$$J2.\text{sum_Torque} = J2.\text{flange_a.tau} + J2.\text{flange_b.tau};$$

For the clutch that has no inertia, the sum_Torque variables are zero as it should be. It is not zero for inertia models J1 and J2 because they can store momentum. Their models include the equation

$$J \cdot a = \text{flange_a.tau} + \text{flange_b.tau};$$

Thus, for inertia we have

$$\text{sum_Torque} = J \cdot a$$

The variable torque.sum_Torque is non zero. For the system, above momentum is not preserved. In reality the drive train is mounted in the car. The chassis provides a corresponding reaction torque, which propagates through the wheels and tires to the road. Thus if we put the drive train above into a chassis model without connecting the bearing connectors of the drive train properly to model the real mounting, we will get wrong simulation results. How can we provide some automatic checks?

For components not storing conserved quantities, we can use `sum_Torque` and the other sum variables and add an assertion that the sum should be zero. For the example, the simulation stops issuing

```
Assertion failed: abs(torque.sum_Torque)< 1E-005
Torque not conserved in the component torque.
```

In the general case, a component needs to include information on what are the storage terms. Tiller and Kittirungsi (2006) propose annotation to be used. For example to indicate that the term $J \cdot a$ above implies storage:

```
Modelica.SIunits.Torque
torqueStorage = J*a
  annotation (storageTerm);
```

In 3 D mechanics forces are vectors and the balances of forces must be set up in the same frame. Moreover, the torque balances are even more complicated since they also include terms referring to the forces acting on the body.

One idea is to add an annotation to

```
Modelica.Mechanics.MultiBody.Interfaces.Frame:
```

```
connector Frame
  "Frame of a mechanical system"
  annotation (ConservedQuantity(
    Force=Frames.resolve1(R.T, f),
    Torque=Frames.resolve1(R.T,t)+
      cross(r,Frames.resolve1(R.T,f))));

  import SI = Modelica.SIunits;
  SI.Position r_0[3];
  Frames.Orientation R;
  flow SI.Force f[3];
  flow SI.Torque t[3];
end Frame;
```

The scope for the right-hand-sides is the local connect (exactly as for e.g. a binding equation in the class) and this takes precedence over the default-summing of quantity-flows to 0. This annotation should only be added once [i.e. for the base-class of all flange-connector and not for each model], and we could alternatively have this built-in in Dymola for this class.

Please, note that if a model component fails to annotate or mark a term as contribution to storage then the check will detect this, i.e., the component model is not conserving properly.

In order not to be forced to model all universe, it is necessary to support infinite sources or sinks for conserved quantities. Again it is possible to use an annotation to mark such components. However, there is a potential risk with ground elements. Assume that we fix the coupled clutch model above by connecting

a ground component to the bearing connector of the component torque. This component can be viewed as a rig where we put the drive train for testing. The rig may be viewed as representing the “infinite mass” of the earth.

Energy conservation is important to check. However, it is more complex. For thermodynamics the heat flows as well as the enthalpy flows are a power flow. However, the energy flow does not always appear explicitly as flow variables in the connectors. Some energy flows can be computed by multiplying the flow quantities by a proper derivative of the corresponding across variable. Examples:

```
pin.v*pin.i           => V*A=W
der(flange.s)*flange.f => m/s*N=W
der(flange.phi)*flange.tau => rad/s*Nm=W
```

It works for Electrical, Rotational and Translational. For MultiBody there is the problem with different coordinate systems.

Establishing energy conservation also includes identification of energy dissipation. For example a resistor “dissipates” energy, or more explicitly, it converts electrical energy into heat. The basic components in Electrical do not include such information.

The automatic checking of conservation may have several objectives. A primary objective is to catch model errors. However, a conservation condition may be violated over time due to numerical drift of the numerical solution for the conserved quantity. This calls for a more sophisticated checking considering the numerical drift. On the other hand it may also be used to improve the numerical solution. A numerical solver may exploit these invariants for automatic selection of tolerances, i.e. the user put tolerances on invariants. Projection methods may be used to numerically control the drift.

Evidently there is a need to include more information in the models in order to be able to perform automatic checking of quantity conservation. Fortunately, quantity storing is done in the basic components such as inertia, body, volume. For energy balances we need also to consider dissipation in for example resistor, damper, pipe friction etc.

6 Conclusions

Dymola’s support of units, unit checking and unit deduction has been described. It has already proven useful. Several errors in the Modelica Standard Library were found. It has encouraged the developers of the Modelica Standard Library to declare vari-

ables representing quantities appropriately. The `displayUnit` concept allows users to enter parameters and plot variables in different units while allowing clean equations without complicating conversion factors because the equations can refer to the quantities in SI units. The `Modelica.Blocks` library includes general blocks to define sources and mathematical operations. Their inputs and outputs have of course no units specified. This may also be the case for some parameters such as gain. At translation of a model for simulation Dymola infers their units in order to improve the variable browsers for entering parameter values and plotting variables during simulation.

Second, the possibilities of checking quantity conservation automatically are discussed. It is in open area where there is a large potential to check that models fulfill the very basic laws of physics including energy conservation, Newton's third law "action equals reaction". Evidently there is a need to include more information in the models to really support automatic checking of quantity conservation and there is a need for extensions of Modelica. Fortunately, it seems as if most of this can be done in the basic components such as inertia, body, volume, capacitor etc which actually stores some quantity. For energy balances it is also necessary to identify and mark dissipation in resistors, friction elements etc.

References

BIPM 2006. The International System of Units (SI), Bureau International des Poids et Mesures, 8th edition, 2006. Available at www.bipm.org/en/si/si_brochure

Dynasim. 2007. Dymola Version 6.1. Dynasim AB, Lund, Sweden. <http://www.dynasim.se/>.

Modelica 2007. Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification, Version 3.0, Available in electronic form at www.modelica.org/documents/ModelicaSpec30.pdf

M.Tiller and B. Kittirungsi: UnitTesting. 2006. A library for Modelica unit testing. Proceedings of the 5th Modelica Conference, Vienna, Austria, 2006, Vol. 2, pp. 695-704.

Balanced Models in Modelica 3.0 for Increased Model Quality

Hans Olsson¹ Martin Otter² Sven Erik Mattsson¹ Hilding Elmqvist¹

¹Dynasim AB, Ideon Science Park, SE-223 70 Lund, Sweden

²German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Oberpfaffenhofen, 82234 Weßling, Germany

{Hans.Olsson, SvenErik.Mattsson, Hilding.Elmqvist}@3ds.com, Martin.Otter@dlr.de

Abstract

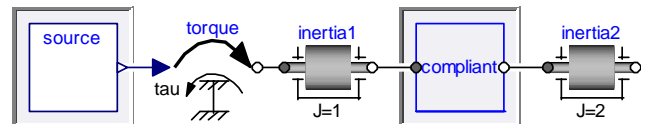
A Modelica model can only be simulated, if the number of unknowns and the number of equations are equal. In Modelica 3.0, restrictions have been introduced into the language, in order that every model must be “locally balanced”, which means that the number of unknowns and equations must match on every hierarchical level. It is then sufficient to check every model locally only once, e.g., all models in a library. Using these models (instantiating and connecting them, redeclaring replaceable models etc.) will then lead to a model where the total number of unknowns and equations are equal. Besides this strong guarantee, it is possible to precisely pinpoint which submodels have too many equations or lack equations in case of error. This paper gives the rationale behind the Modelica 3.0 design choices including proofs of the new guarantees, and discusses the limitations of this approach.

1 Background

In a causal modeling paradigm, where only input/output blocks are used, it is straightforward to verify that all input connectors have been connected, and thus causal modeling naturally lead to a simple plug and play metaphor for end-users. The goal is to ensure that acausal Modelica model components are as convenient to use for end-users.

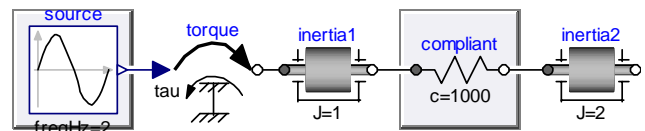
The need for this is growing in importance with larger and more complex model libraries and with companies expanding Modelica usage from research to development. Furthermore, libraries with template models, i.e., incomplete models with replaceable components, like VehicleInterfaces, PowerTrain, VehicleDynamics library, can easily lead to wrong models in Modelica 2 when using the templates, without being able to give reasonable diagnostics for the source of the error. Shortened production cycles

imply that we want to verify correctness early, in particular already for incomplete models where implementation of the parts is left open. An example below (sunken icons means partial replaceable components) shows a driveline where the input torque and compliance between the inertias are unspecified.



The goal is that by separately verifying that the template model is correct, and imposing restrictions on the models we plug in, we can be certain that the complete model is correct.

Without the restrictions, the tool would need to perform a global analysis, and if the complete model is not balanced we would not know whether the implementation of the part or the template model itself was in error. Having to verify this for all combinations of sources and gears (one is shown below) is not practical:



The first possibility would be to improve the analysis of structural singularities for Modelica 2 to find the errors without requiring balancing, but instead use other information including annotations and confidence in different equations [2]. Similar techniques are also useful at the lowest level to go from one equation too many in the current model to pinpointing which equation is superfluous.

Previously there have been checks in Dymola [3] (since Dymola 5.3 released in 2003) to determine in case of an unbalanced simulation model which submodels are incorrect based on the actual use. That was introduced to help users in finding errors, but it did not always work satisfactorily since it was not always possible to determine how many equations should be present in each model (the best what could be done in general was to determine a range for the

number of equations); thus for the complete model it was not possible to determine whether the error was in the template or in one of the implementations – and if so which one. Thus without stricter language rules no reliable diagnostics could be given to users to pin-point the errors.

A related work [4] allows unbalanced classes, maintains the (un)balancing of connectors and models when modified. The difference is that in Modelica 3.0, it is enforced that models are balanced from the start and on all levels.

2 Number of equations in the model

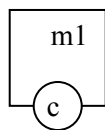
In order to verify whether a component model is balanced we must have a clear definition of how many equations the model contains – and how many equations it should contain (both based on its interface).

2.1 Restrictions on physical connectors

The goal is that combining models having (physical) connectors and connecting them (in legal ways) we should get new balanced models; without imposing additional restrictions on the models or requiring adding equations.

Consider the simplest case of a model where the only public part is one connector, and it is “physical”, i.e. containing n_f flow variables “Real f [nf]” and n_p non-causal, non-flow, “potential” variables “Real p [np]” (i.e., no connector variable has the input or output prefix); and the model is balanced, i.e., the model requires that externally a specific number of equations (n_e) is provided, in order that all unknowns of the model can be uniquely computed together with the internal equations in the model. We will call these required equations in the sequel external equations of a model component. The number of external equations has to be uniquely defined by the interface of the model. The balancing is that the number of unknown variables equals the number of equations defined inside the model plus the number of external equations.

The simplest use of a model m_1 with connector c is that the connector is unconnected.



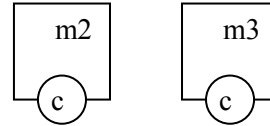
The Modelica semantics does state that the flow-variables are summed to zero, whereas the “potential” variables should be equal [1].

This leads to the external equations

$$m1.c.f = 0; \quad // \text{ nf equations}$$

Since m_1 requires n_e external equations and instantiating the component gives n_f equations, we have the requirement: $n_e = n_f$.

The next case is to have two of such components not being connected:



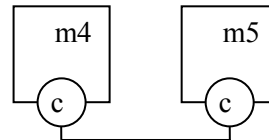
In this case we get the external equations

$$m2.c.f = 0; \quad // \text{ nf equations}$$

$$m3.c.f = 0; \quad // \text{ nf equations}$$

Since m_2 requires n_e equations and m_3 requires n_e equations, we have the requirement: $2 \cdot n_e = 2 \cdot n_f$

The next case is to have two of such components, but being connected:



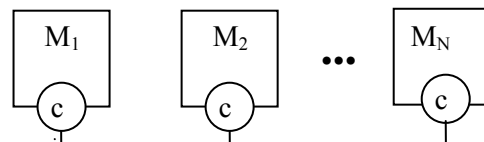
Here we get connection equations:

$$m4.c.f + m5.c.f = 0; \quad // \text{ nf equations}$$

$$m4.c.p = m5.c.p; \quad // \text{ np equations}$$

Since m_4 requires n_e equations and m_5 requires n_e equations, we have the requirement: $2 \cdot n_e = n_f + n_p$

The final case is to have N components that are connected together:



Here we get one time the zero-sum equations for the flow variables and $N-1$ identity equations for the potential variables. Since every model requires n_e equations, we have the requirement:

$$N \cdot n_e = n_f + (N-1) \cdot n_p$$

To summarize, we get the following relations that all have to be fulfilled, in order that instantiating and optionally connecting components does not require to add any more equations (= necessary and sufficient conditions):

$$1^{\text{st}} \text{ model (m1):} \quad n_e = n_f$$

$$2^{\text{nd}} \text{ model (m2,m3):} \quad 2n_e = 2n_f$$

$$3^{\text{rd}} \text{ model (m4, m5)} \quad 2n_e = n_f + n_p$$

$$4^{\text{th}} \text{ model (M}_1, \dots, M_n) \quad N \cdot n_e = n_f + (N-1) \cdot n_p$$

or equivalently

$$n_f = n_e$$

$$n_p = n_f$$

This leads to the conclusion that the number of flow and non-causal, non-flow variables must match (counting arrays as the number of elements of simple types), and this must correspond to the number of external equations for this connector.

Models may also have variables that are declared with the input prefix, both in a declaration of a model and in a (top-level) connector. These variables are treated as unknowns in a model. It is natural to require that for all these input variables external equations must be provided.

In order to force the user that all missing equations for a model are provided when instantiating the model, it is required that all input variables declared in a model are provided as modifiers and that all inputs in a (top level) connector are provided by connecting the connector. Since a connector must have the same number of flow and potential variables (see derivation above), the means that a connector with an input variable A must be connected to another connector where variable A has the output prefix (an exception of this last rule will be discussed in the next section).

According to these rules, it is no longer allowed to provide modifiers for other variables (with exception of variables declared with the input, parameter or constant prefix) or add other equations for the component externally, because all of these actions would introduce superfluous equations.

To summarize, we have basically the following requirements (for simplicity, not yet considering special cases such as over-determined connectors, non-causal variables with a declaration equation, partial models, or connectors with input variables that are not connected):

1. The number of flow variables in a connector must be identical to the number of non-causal, non-flow variables (variables that do not have a flow, input, output, parameter, constant prefix).
2. The number of equations in a model = number of unknowns – number of inputs – number of flow variables (of top-level public connector components). For the equation count, components are not taken into account, because this is taken into account by the next rule 3.
3. When using a model, i.e., making an instance, all missing equations of this component must be provided to make the component “balanced” by:

- a) Connecting connectors or by leaving “physical” connectors unconnected (since the missing equations are then automatically introduced by setting all flow variables to zero).
- b) Providing a modifier for every non-connector component variable with an input prefix. Besides parameters and constants, modifiers on other variables are no longer allowed.

The above rules shall be clarified with a few simple examples (assuming a global definition

```
import SI=Modelica.SIunits;

connector FluidPortA
  SI.Pressure          p;
  flow SI.MassFlowRate m_flow;
  input SI.SpecificEnthalpy h_inflow;
  output SI.SpecificEnthalpy h_outflow;
end FluidPortA;

connector FluidPortB
  SI.Pressure          p;
  flow MassFlowRate    m_flow;
  output SI.SpecificEnthalpy h_inflow;
  input SI.SpecificEnthalpy h_outflow;
end FluidPortB;
```

The two connectors FluidPortA and FluidPortB are valid, since they each have 1 flow and 1 non-causal, non-flow variable and 2 causal variables. Note, whenever input/output prefixes are present, there are connection restrictions because the block diagram semantics holds (e.g. an output cannot be connected to an output). As a result FluidPortA can only be connected to one FluidPortB, but not to another FluidPortA.

```
connector WrongFlange // wrong connector
  SI.Angle          angle;
  SI.AngularVelocity speed;
  flow SI.Torque    torque;
end WrongFlange;
```

This connector is not valid, since the number of flow and non-flow variables is not the same. This is a typical situation of “old” connectors, such as the connectors of the (obsolete) ModelicaAdditions.MultiBody library. Both these “old” connectors, as well as the “WrongFlange” connector above can be made valid, by using the prefix input or output for one of the non-flow variables (similarly to FluidPortA and FluidPortB above).

```
model Pin
  SI.Voltage v;
  flow SI.Current i;
end Pin;

model Capacitor
  parameter SI.Capacitance C;
  SI.Voltage u;
  Pin p, n;
equation
  0 = p.i + n.i;
  u = p.v - n.v;
```

```

    C*der(u) = p.i;
end Capacitor;

```

The Capacitor model has 5 unknowns¹ (u , $p.v$, $p.i$, $n.v$, $n.i$) and 2 flow variables ($p.i$, $n.i$). It is therefore required that this model has $5 - 2 = 3$ equations, and the model fulfills this requirement.

```

model Test1
  Capacitor C1(C=1e-6); // o.k
  Capacitor C2(u=sin(time)); // wrong
end Test1;

```

The declaration of C1 is correct, because a modifier for a parameter is given. The declaration of C2 is wrong, because it is no longer allowed in Modelica 3 to provide a modifier for a variable that does not have a constant, parameter or input prefix.

```

model VoltageSource
  input SI.Voltage u;
  Pin p, n;
equation
  u = p.v - n.v;
  0 = p.i + n.i;
end VoltageSource;

```

The VoltageSource model has 5 unknowns (u , $p.v$, $p.i$, $n.v$, $n.i$), 2 flow variables ($p.i$, $n.i$) and 1 input variable (u). It is therefore required that this model has $5 - 2 - 1 = 2$ equations and the model fulfills this requirement.

```

model Test2
  ...
  VoltageSource V1(u=sin(time)); // o.k
  VoltageSource V2; // wrong
  ...
end Test2;

```

Component V1 is correct, because the missing external equation for the unknown input u is given as modifier.

Component V2 is not correct, because no modifier or equation is provided for the missing unknown input “ u ”.

The counting for non-connector inputs (such as u) is defined as if they always had a declaration equation. Thus the result would be the same for this modified model:

```

model VoltageSource
  input SI.Voltage u=0; // Default
  Pin p, n;
equation
  u = p.v - n.v;
  0 = p.i + n.i;
end VoltageSource;

```

¹ Alternatively, one could define „ u “ as known (because it is a potential state) and „ $der(u)$ “ as unknown. However, this does not hold in general, since $der(\cdot)$ might have an expression as argument. For this reason, $der(\cdot)$ is used as operator that does not have an influence on the equation counting.

This implies that we can add default values without modifying the use of the model.

2.2 Correlations and non-connector inputs

Causal variables are not limited to connectors, but there are also non-connector inputs and outputs – which can be viewed as “time-dependent parameters”. The non-connector outputs have no special significance here (they are useful to indicate special interesting variables). The non-connector inputs are for the balancing always counted as having a binding equation – and must have a binding equation in the complete model. This simplifies the requirement for counting equations such that modifiers are not counted as providing external equations for the model; since they are seen as replacing old declaration equations with new ones of similar size. The important aspect is that other alternatives, such as giving normal equations for them – or modifying some non-input would not preserve the balancing of equations.

One example demonstrating this issue is correlations; i.e., relations constraining a set of variables to be on a hyper-plane of a certain dimension. The simplest is a correlation involving two variables; in this case the variables will simple be on a curve. We can arbitrarily declare one as input, but the correlation normally is written as just an equation relating the variables (and the line could have straight segments in both x and y direction). *Note: This example is illegal in Modelica 3.0.*

```

partial model Correlation
  input Real x;
  Real y;
end Correlation;

model UseCorrelation
  // Wrong in Modelica 3
  replaceable Correlation corr;
equation
  corr.y=2+time;
  /* Same number of equations as
   * modifying "x"; could also be
   * written as modifier for y */
end UseCorrelation;

model LineCorrelation
  extends Correlation(x=3);
equation
  x+y=0;
end LineCorrelation;

model Complete=UseCorrelation
  (redeclare LineCorrelation corr);
// model is not balanced since
// 2 unknowns (x,y), but 3 equations:
// x + y = 0;
// x = 3;
// y = 2 + time;

```


In practice we would have a set of correlations and a set of uses of them, and what we want to verify is that they are all correct without performing all the tests (which would lead to a combinatorial explosion in the number of tests). For Modelica 3.0 we wanted to support the model `Correlation` and the use in `LineCorrelation` and `Complete` – without the possibility of too many equations in the `Complete` model. Since we would expect these models to be developed by separate teams (and normally be part of larger systems) it represents exactly the situation to avoid – an unbalancing due to the interaction of several correct models – and without a clear description of where the error is. The solution is in this case to disallow the construct in `UseCorrelation` for non-connector inputs, and find another way of providing the correlations: Component “`corr`” in `UseCorrelation` has one input variable and it is required to provide a modifier for this variable in order to make model “`corr`” balanced. This is not the case above and therefore the model is not correct. This restriction on modifiers to parameters, and non-connector inputs is part of Modelica 3.0.

The ideal solution for modeling the correlation would be that `UseCorrelation`, `LineCorrelation` (except for ‘`x=3`’), and `Complete` would all be legal, and rewrite `Correlation` to allow this. The first attempt was to have some way to disable the balancing test for `Correlation` and derived classes; that implied that only the class `Complete` could be checked; and in case it failed it would be impossible to determine whether `UseCorrelation` or `LineCorrelation` should be modified. During the design of Modelica 3.0 several attempts were made of introducing a special syntax for stating that `Correlation` is lacking a certain number of equations – without defining ‘`x`’ as an input (because either `x` or `y` shall be defined when using the component). This requires the introduction of additional non-intuitive syntax and the final decision was to change instead ‘`x`’ to a connector input and modify the language rules to allow unconnected connector inputs and provide the binding equation for the input connector as equation. The example then becomes (the differences in `Correlation` are highlighted):

```

partial model Correlation
  InputReal x;
  Real y;
  connector InputReal=input Real;
end Correlation;

model UseCorrelation
  replaceable Correlation corr;
equation
  corr.y=2+time;
end UseCorrelation;

```

```

model LineCorrelation
  extends Correlation;
equation
  x+y=0;
end LineCorrelation;

```

```

model Complete=UseCorrelation
  (redeclare LineCorrelation corr);

```

In this case `LineCorrelation` may not use modifiers for ‘`x`’, since ‘`x`’ is a connector, and we are thus once more certain that the number of equations will automatically balance. This is used for `Modelica.Media`, and can be used in other cases for correlations as well.

Note that `UseCorrelation` is exactly identical to the original version, but is now legal due to the change in the `Correlation` model (i.e. `corr.y = 2 + time`, is the missing equation for the input connector `x`).

This approach was decided upon even though it has the disadvantage that we allow unconnected input connectors, and to count equations we thus have to combine the normal equations and the equations for missing input connections in the count of equation. In this case ‘`corr.x`’ is not connected in the `UseCorrelation` model; and instead a non-connect equation is giving.

This can be compared to a causal paradigm, where we would just require that ‘`corr.x`’ must be connected. However, even if the use above is legal a tool could still inform the user that ‘`corr.x`’ lacks a connection if `UseCorrelation` or `LineCorrelation` are not balanced or if the simulation model is structurally singular, in order to help in pinpointing the error.

3 Locally balanced models

In the previous chapter, the counting rules have been sketched for the most important cases. We will now formulate the exact rules and what guarantee can be given:

A model or block is called “locally balanced” if the local number of unknowns matches the local equation size (both terms are defined below). Note, that all counts are performed after expanding all records and arrays to a set of scalars of primitive types. We will here ignore inner and outer components, as well as over-determined connectors, to simplify the definitions and results – for complete definitions see the Modelica 3.0 specification [1].

The local number of unknowns is the sum of:

- For each declared component of specialized class type (Real, Integer, String, Boolean, enumeration and arrays of those, etc) or record it is the “number of unknown variables” inside it (i.e., excluding parameters and constants).
- For each declared connector component, it is the “number of unknown variables” inside it (i.e., excluding parameters and constants).
- For each declared block or model component, it is the “sum of the number of inputs and flow variables” in the (top level) public connector components of these components.

The local equation size is the sum of:

- The number of equations defined locally (i.e. not in any model or block component), including modifier equations, and equations generated from connect-equations.
- The number of input and flow-variables present in each (top-level) public connector component, i.e. the externally needed equations.
- The number of (top level) public input variables that neither are connectors nor have binding equations, i.e., further externally needed equations.

The following restrictions are imposed in Modelica 3.0

- **All non-partial model and block classes must be locally balanced.**
- In a non-partial model or block, all non-connector inputs of model or block components must have binding equations (i.e. they are defined in a modifier).
- Modifiers for components shall only contain re-declarations of replaceable elements and binding equations for parameters, constants (that do not yet have binding equations), inputs and variables having a default binding equation.
- In a connect-equation the primitive components of the two connectors must have the same primitive types, and flow-variables may only connect to other flow-variables, causal variables (input/output) only to causal variables (input/output).
- A connection set of causal variables (input/output) may at most contain one inside output connector or one public outside input connector. [i.e., a connection set may at most contain one source of a signal, which is the “usual” semantics for block diagrams.]
- At least one of the following must hold for a connection set containing causal variables:

- (1) the model or block is partial,
- (2) the connection set includes variables from an outside public expandable connector,
- (3) the set contains protected outside connectors,
- (4) it contains one inside output connector, or
- (5) one public outside input connector, or
- (6) the set is comprised solely of an inside input connector that is not part of an expandable connector.

i.e., a connection set must – unless the model or block is partial – contain one source of a signal (the last items covers the case where the input connector of the block is unconnected and the source is given as equation in the equation or algorithm section).

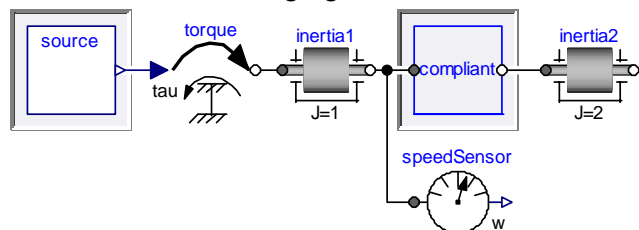
- A protected outside connector must be part of a connection set containing at least one inside connector or one declared public outside connector (i.e., it may not be an implicitly defined part of an expandable connector).

4 Plug and play

We will show that if a user uses locally balanced classes and follow the language restrictions and drags and drop components and connect them, they will automatically build locally balanced classes as shown below. We will go through this starting from an even more restricted case; in the conclusion we will explain why these rules are not present in the language.

4.1 Only components and connections

Assume we build a non-partial model (or block) composed solely of components of model and block classes (with optional legal value modifiers applied) and connections that satisfy all restrictions, as it is the case in the following figure:



Furthermore for connection sets involving causal variables the connection set should satisfy case 4 in the itemized lists above (=contain an inside output connector generating the signal) – i.e. explicitly excluding case 6 (since cases 1, 2, 3, and 5 cannot ap-

ply here). The model or block is then automatically locally balanced.

Note: The excluded case (6) would correspond to removing the source-component above, and instead write a textual equation for torque.tau. This also applies to case (3), which is less needed.

In this case the local number of unknowns corresponds to the number of inputs and flow variables in the public connectors of the components; and the equations to the equations generated by connection equations.

We can split the connectors into causal and non-causal parts (due to the restriction that connection sets may not mix the two; this restriction was added in Modelica 3.0 to allow this analysis).

For the causal part we have the local number of unknowns corresponding to the number of inputs in the public connectors of the components. Among these variables we have n_i inputs and n_o outputs, and the number of equations is thus $n_i + n_o - 1$; the case 4 above gives $n_o = 1$ yielding the local number of equations $n_i + n_o - 1 = n_i + 1 - 1 = n_i$ exactly matching the local number of unknowns.

For the non-causal part we have the local number of unknowns corresponding to the number of flow variables in the public connectors. Assume there are n connectors in this set, and each connector has n_f flow-variables and $n_p = n_f$ non-causal, non-flow variables (“potential variables”). We have $n \cdot n_f$ local number of unknowns; one zero-sum equation for flow variables and $n - 1$ equality equations for the potential variables; in total this gives the size

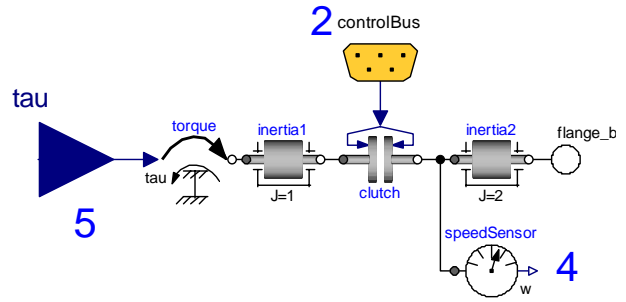
$$n_f + (n - 1) \cdot n_p = n_f + (n - 1) \cdot n_f = n \cdot n_f$$

which exactly matches the local number of unknowns.

This case is important for users combining models from different libraries – and ensures that as long as the user only combines correct models without introducing simple variables or equations, the model is automatically balanced.

4.2 Connectors

Assume we extend the above list to include components of connectors classes (without any value modifiers), that cases 3 and 6 in the itemized list above does not apply, and that each connector component is connected (case 1 does still not apply since we build a non-partial model or block). The cases are indicated below.



We will apply the split into causal and non-causal part. For the non-causal part it seems that the previous proof goes through automatically – this is true with one minor caveat: if a (public) outside connector had not been connected (the case we excluded) it would *not* have been part of a connection set and would have given 0 equations instead of the correct number n_f .

For the causal part it is more complex since we have both protected and public causal variables. If we disregard case expandable connectors and use superscript i/o for inside/outside, and subscript n for nodes (regardless of input/output).

The unknowns are given by the local connectors: n_i^o outside inputs, n_o^o outside outputs, n_n nodes (protected connectors), and the subcomponents: n_i^i inside inputs. The number of equations ($n-1$) is extended with n_i^o outside inputs. Thus for balancing we get the requirement:

$$n_i^o + n_o^o + n_n + n_i^i = (n_n + n_i^o + n_o^o + n_i^i + n_o^i - 1) + n_i^o$$

Simple cancellations gives: $n_i^o + n_o^i = 1$, or stated differently: either case (5) an outside input connector, or case (4) an inside output connector. The either is due to the restriction about multiple sources in a connection set. For expandable connectors (case 2) the same rules apply after we have deduced the causality; this will also influence the number of unknowns.

4.3 Redeclare of components

When redeclaring a component, the missing equations for the component must be either provided via modifier equations (parameters, inputs) or connectors must be connected. When these restrictions are fulfilled, the redeclared component is automatically locally balanced.

One situation has to be treated specially: If the redeclared component introduces additional connectors that are not defined in the constraining clause. Unless the connectors are part of a redeclared inherited top-level component, it is not possible that a user

can connect to these newly introduced connectors. This is uncritical, if the not connected connectors do not have input variables, since the default connection semantic will set the flow variables to zero. Consequently, the restriction is introduced in Modelica 3.0 that additional connectors that are not defined in the constraining clause are default connectable, i.e. shall not have input variables.

A record or connector component that is directly replaceable or more commonly declared using a connector from a replaceable package also has a parameter dependent size. In such a case a redeclaration may add additional unknowns – which should also be balanced with matching equations. This is a “parameter-dependent size” and can be handled using the techniques in the next section – except that only unknowns are added in this way – but no equations, and at first it seems that this will inevitably lead to unbalanced models. However, it is possible to handle this correctly: by not using replaceable connectors directly, but instead use a replaceable package containing connectors and corresponding models (or functions) – similarly as in the Modelica.Media package.

This is a special case and we will not discuss the details. However, it has a direct relevance to a more fundamental change introduced in Modelica 3.0. Previously a connector component of a replaceable model component was implicitly replaceable, i.e. the problem that a `redeclare` could introduce missing equations was present for any replaceable model component having connectors – even if the connectors were not replaceable.

5 Parameter dependent sizes

An important aspect of the counting of equations is that it holds not only for the current set of parameters, but for any legal set of parameters values. The restriction in Modelica 3.0 is formulated such that even though the model should be balanced in all cases, the tool does not have to verify this. The reason was that *at the time* it was not possible to verify that a given set of restrictions ensures that sub-models will always be within the restrictions, *and* that user libraries could be rewritten to conform to this.

Dymola can perform this test in several cases as will be outlined here; and in the remaining cases it is verified for the actual parameter values and a warning given.

The number of scalar variables is obtained by recursively symbolically adding the number of components of each variable:

- A scalar variable has the size 1.
- An array $v[n]$ has the size: $n \cdot \langle \text{the size of its elements} \rangle$. Modelica implicitly assumes that $n \geq 0$. A multidimensional array is in Modelica considered as a nested array. For example, a matrix $M[m, n]$ has the size $m \cdot n \cdot \langle \text{the size of its elements} \rangle$. If the size is declared using the colon operator, $v[:] = \dots$, the size is represented as $\text{size}(v, 1) \cdot \langle \text{the size of its elements} \rangle$. The idea is to represent the size expressions of arrays symbolically as defined by the model developers.
- The size of a record is the sum of the size of its components.

The current restrictions when counting variables are evaluation of sizes of arrays of components and the conditions of conditional components.

The number of components of an equation is counted by traversing all its subexpressions and deducing the dimensionality and the size of each dimension and propagating this information upwards without any evaluation. At the top level the number of components is formed in a way analogous to that of variables. Also size constraints are collected for immediate or deferred checking. An interesting fact is that the size of a for-loop equation can be formed as the sum of the elements of an array constructor. One restriction is that the instantiation procedure may have evaluated some conditions.

The comparison of the number of variables and equations is done in 3 steps:

- First all variables which bindings cannot be modified (`protect`, `final`, `constant`) are substituted symbolically. If Dymola can symbolically deduce that the problem is balanced the check was successful in this respect. It means that the model is balanced irrespective of how a user rebinds or sets parameters that may be rebound or set.
- Otherwise, Dymola substitutes all non-literal bindings. If Dymola now can show that the problem is balanced, the comparison is finished. It means that the user can change parameter values that are literals, but not otherwise rebind parameter values without risking making the problem non-balanced. A remedy is to define critical parameter bindings to be `final`.

- The third step is to force evaluation of all size parameters and then compare. This is what Dymola has done previously when checking or translating a model.

As an example consider model `Modelica.Blocks.Continuous.StateSpace`. The essence is:

```

block StateSpace
  parameter Real A[:, size(A,1)];
  parameter Real B[size(A,1), :];
  parameter Real C[:, size(A,1)];
  parameter Real D[size(C,1), size(B,2)]
    =zeros(size(C, 1), size(B, 2));
  extends Interfaces.MIMO(
    final nin= size(B, 2),
    final nout=size(C, 1));
  output Real x[size(A, 1)];
equation
  der(x) = A*x + B*u;
  y = C*x + D*u;
end StateSpace;

```

Checking the model in Dymola 7.0 results in

```

Model having the same number of
unknowns and equations:
  size(A, 1) + size(B, 2) + size(C, 1)

```

The counting of the unknown variables which are x , u and y , gives

```
size(A, 1) + nin + nout
```

The bindings for the parameters `nin` and `nout` are `final` and can be used for substitution, which gives the logged result. The counting of equations gives first `nin` for the inputs. The size check of

```
der(x) = A*x + B*u;
```

has to check possible size constraints for each subexpression. First, the matrix-vector multiplication, $A*x$ requires $\text{size}(A,2)=\text{size}(x,1)$. Exploring the declarations of A and x shows that either side is equal to $\text{size}(A,1)$. The product $A*x$ is a vector with $\text{size}(A*x,1)=\text{size}(A,1)$. The product $B*u$ requires $\text{size}(B,2)=\text{size}(u,1)$. Exploring the declaration of u gives $\text{size}(u,1)=\text{nin}$ and the final binding to `nin` gives $\text{nin}=\text{size}(B,2)$. Thus the constraint is fulfilled. The product $B*u$ is a vector with $\text{size}(B*u,1)=\text{size}(B,1)$. Next, the sizes of the two terms $A*x$ and $B*u$ must be equal. They are both vectors and the size constraint is $\text{size}(A*x,1)=\text{size}(B*u,1)$. Since $\text{size}(A*x,1)=\text{size}(A,1)$ and $\text{size}(B*u,1)=\text{size}(B,1)$ and since the declaration states $\text{size}(B,1)=\text{size}(A,1)$, the constraint is fulfilled. The resulting sum is a vector of $\text{size}(A,1)$. Since the declaration of x specifies $\text{size}(x,1)=\text{size}(A,1)$ all the size constraints of the equation are fulfilled symbolically for all allowed A and B matrices and it has $\text{size}(A,1)$ components. Similarly the equation $y = C*x + D*u$ is type consistent and has $\text{size}(C,1)$ equations.

Dymola's facility for checking that two symbolic expressions are equal is rather elaborate. However, it cannot handle all cases such as complicated `for`-loop equations where there are, for example, conditions on the loop iterator. Dymola then resorts to numerical evaluation.

6 Limitations of the approach

The previous section shows that the rules in Modelica 3.0 make it possible to provide early checks of models that will avoid several hard to find errors when completing large models. The early checks are possible, since we only need the interface of sub-components. However, some errors are still possible when assembling sub-models and the natural question is why these errors cannot be handled in a better way.

6.1 Why are not all restrictions in the language?

As noted above we can prove that models are automatically balanced if built subject to certain restrictions, but not all of these restrictions are part of the language. This might seem odd considering that we want to ensure correctness early on, but it is necessary to allow textual (non-connector) equations to be given for low-level models. However, the check can still be performed at the same level – and uses the same description of balanced models; the only difference is that if the guidelines are followed the check becomes even simpler. When the guidelines are not followed, a user would have to provide non-connector equations as a replacement for connections; thus for a model with only connector equations the simpler restrictions hold. This makes it straightforward to provide good diagnostics, while preserving the low-level openness of Modelica. Furthermore, a general recommendation is to avoid mixing connections and textual equations (see e.g. [5]); which makes it easier to separate the two cases.

The examples where this is necessary include writing basic models such as a resistor where equations are given for the connectors - instead of adding additional connections, and correlations for media-models are built such that there are multiple potential inputs (see section 2.2 Correlations and non-connector inputs). Allowing equations for input connectors is also convenient in some other cases (e.g. when using table-lookup blocks); and by having the semantics above we avoid introducing a special se-

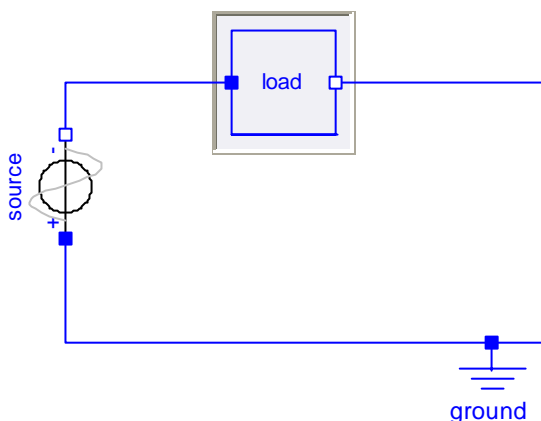
semantic construct for defining the number of external equations needed for the media-models.

6.2 Requirements beyond balanced?

Having a balanced model is only a necessary requirement to be able to simulate it, but it is not sufficient. Whether a non-linear system of differential-algebraic equations has a solution is NP-hard in general; imposing restrictions to ensure a solution would impose such strict rules on the modeling (such as convex equations) that it would be not practical in general.

For the complete model a strong requirement is that the system of equations is structurally regular (i.e. no singularity when looking at the structure and ignoring the actual values). Dymola can also perform this check already for incomplete models and then use a generic coupling for local (replaceable) components, and for top-level connectors. This can provide useful diagnostics for many cases.

However, structural regularity is not entire well-defined, e.g. Dymola actually uses ± 1 from connections as well as zeros (finding more errors), and in contrast to balanced models structurally singular does not provide strong guarantees. Even plugging in parameters might turn a structurally regular system into a structurally singular one, since structurally singular only ensures that the equations are non-singular for “most” values of the non-zero elements. Obviously this also applies to redeclarations, especially since one often uses simplified models as test-cases. A simple example would be an electrical circuit testing different components with an ideal source:



This model (with an ideal voltage source) will become structurally singular if we plug in a short-circuit component as a load.

6.3 Restrictions on partial models?

Currently there are no balancing restrictions on partial models in Modelica, and the contents of base-classes are expanded prior to verifying the balancing. This is the formal semantics; a tool may internally handle this in a better way, taking special care of the non-trivial handling of connection sets, and of multiple inheritance of the same component.

The reason for the lack of restrictions is that the number of equations needed in derived models depend on whether the partial model is just an interface (e.g. TwoFlanges in the Rotational library – this just has two flange connectors), or contains an incomplete set of equations (e.g. Rigid in the Rotational library – which also specifies that the angles are identical). If we compare with e.g. Java this implies that a partial class may be either an interface or an abstract class.

A possible extension would be to have separate keyword for pure interfaces, and restricted such that only public connectors, parameters, and causal variables are present. The number of equations needed in derived classes would be uniquely defined from the interface. In that case it would not be necessary to verify that the interface is “balanced”, since it would follow automatically from the requirement on interfaces, and on connector classes.

A practical smaller extension would be to require that partial models may only be locally underbalanced, i.e. lacking equations, but not have too many equations.

These possible extensions have not yet been investigated in details.

7 Conclusions

The new restrictions in Modelica 3.0 make it possible to provide diagnostics earlier in the development process, while still maintaining the low-level openness of Modelica. These early diagnostics both shortens development time, and makes it possible to provide an interface for end-users where certain errors cannot occur – thus reducing the deployment and training cost for these users.

In Dymola 7.0, the restrictions introduced in Modelica 3.0 are supported, but are only imposed when using the Modelica Standard Library 3 (or later). This allows users to continue to run correct Modelica 2 models.

References

- [1] **Modelica 3.0 Language Specification**, Modelica Association, September 2007.
<http://www.modelica.org/documents/ModelicaSpec30.pdf>
- [2] P. Bonus, P. Fritzson. **Automated Static Analysis of Equation-Based Components**. SIMULATION: Transactions of the Society for Modeling and Simulation International. Special issue on Component-based Modeling & Simulation. Vol 80:8, 2004.
- [3] **Dymola**, by Dynasim AB, Sweden. See www.dynasim.se for more information.
- [4] D. Broman, K. Nyström, P. Fritzson: **Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta**. In Proceedings of the 5th international conference on Generative programming and component engineering
- [5] M. Tiller: **Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications**. In Proceedings of Modelica'2003 conference.
http://www.modelica.org/events/Conference2003/papers/h31_parser_Tiller.pdf

Appendix – Over-determined connectors

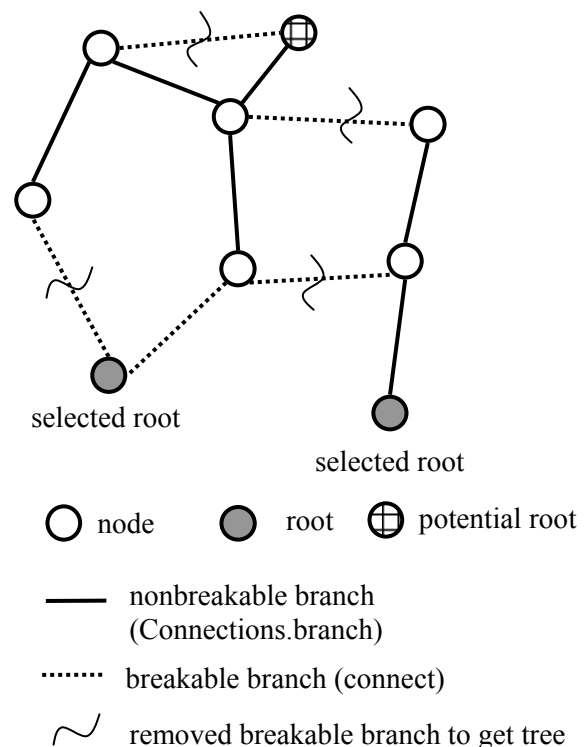
Over-determined connectors have been introduced in Modelica 2.1 to handle a certain class of consistently over-determined set of differential-algebraic equations, for example 3-dim. mechanical systems: Since a MultiBody connector contains the transformation matrix between the world frame and the connector frame, and there are constraints between the elements of a transformation matrix, connecting components with such a connector can lead to an over-determined (but consistent) set of unbalanced equations that have a mathematically well-defined solution. The over-determined connectors are defined and used in such a way, that a Modelica tool is able to remove the superfluous (consistent) equations arriving at a balanced set of equations, based on a graph analysis of the connection structure. For equation counting, it is of course important to take this special treatment into account:

A type class with an equalityConstraint(..) function declaration is called over-determined type. A record class with an equalityConstraint(..) function definition is called over-determined record. The equalityConstraint(R1,R2) functions are used to define the minimal number of equations stating that over-determined types or records R1 and R2 are identical.

A connector that contains instances of over-determined type and/or record classes is called over-determined connector.

Every instance R_i of an over-determined type or record in an over-determined connector is a node in a virtual connection graph that is used to determine when the standard equation " $R1 = R2$ " or when the equation " $0 = \text{equalityConstraint}(R1,R2)$ " has to be used for the generation of connect(...) equations. The branches of the virtual connection graph are implicitly defined by "connect(..)" and explicitly by Connections.branch(...) statements. Additionally, corresponding nodes of the virtual connection graph have to be defined as roots or as potential roots with built-in functions Connections.root(...) and Connections.potentialRoot(...), respectively. Connections are treated as "breakable" branches. By removing appropriate breakable branches, the virtual connection graph is transformed into a set of spanning trees, each comprised of one root.

An example is given in the figure below, where all "dotted" lines characterize "connect(...)" equations. After building up the spanning trees, the connections that have to be removed to arrive at a spanning tree, are specially handled for the generation of the connection equations (see below):



For potential roots the model tests if the root is selected, and then uses different equations. The flow-variables always give the same equations as normal connections, but for "potential" (=non-flow) variables this is different: If a connect(..) equation is

not “broken”, the standard equality equations hold. If a `connect(..)` equation is marked as “removed” in the virtual connection graph, less equations are provided by using the residual equations defined by the type or record specific function `equalityConstraint()` (shorted to `r()` below – with number of equations n_r) taking the two “potential” variables of the connected connectors as input arguments.

If we examine the same cases as in Figure 1 and consider `m1.c`, `m2.c`, `m3.c` as unconditional roots we get the same result for the left and for the right-case (the connection must be a removed branch since two unconditional roots are connected):

```
0 = m2.c.f + m3.c.f; // nf equations
0 = r(m2.c.p, m3.c.p); // nr equations
```

and thus we get:

$$\text{left model (m1):} \quad n_e^{\text{root}} = n_f$$

$$\text{right model (m2,m3):} \quad 2n_e^{\text{root}} = n_f + n_r$$

or

$$n_e^{\text{root}} = n_f$$

$$n_r = n_f$$

If we instead have a potential root, we will in the left (unconnected) case select a root. If connected to a similar component one of them will get a root and the other one not, and the connection will not be broken (i.e. normal connection equations are introduced). We then get the size-constraint:

$$\text{left model (m1):} \quad n_e^{\text{root}} = n_f$$

$$\text{right model (m2,m3):} \quad n_e^{\text{root}} + n_e^{\text{non-root}} = n_f + n_p$$

Combing the two cases results in:

$$n_e^{\text{root}} = n_f$$

$$n_r = n_f$$

$$n_e^{\text{non-root}} = n_p$$

If there are several potential roots they should all give the same external equation count. The requirement on the residual size ($n_r = n_f$) is included in the Modelica specification [1], but additionally only the rooted external equation count is included ($n_e^{\text{root}} = n_f$) and not the non-rooted equation count.

Thus the balancing rules in Modelica 3.0 for over-determined connectors are incomplete and hold only for a very special case (e.g. when a MultiBody component is directly connected to the world object, which is a definite “root” of the virtual connection graph). This should be corrected in a future revision of the Modelica Specification. In the remaining part

of this section, the non-rooted equation count is also taken into account.

In a similar way as in section 2.1 “Restrictions on physical connectors”, the above derivation is also formulated in form of requirements on connectors and models (the derivation requires extending the above test models with a mixture of normal potential variables and potential variables of over-determined records or types):

1. The number of flow variables in a connector must be equal to the number of (normal) non-causal, non-flow variables + the number of residual equations of over-determined records and types (in the set of non-flow variables, the over-determined records and types are not included, because they are included via the residual equations)
2. The number of equations in a model = number of unknowns
 - number of inputs
 - number of flow variables
 - ((for every **Connections.Branch(R1,R2)**) and (for every **Connections.potentialRoot(R1,...)** where **Connections.isRoot(R1) = false**):
 - number of R1 variables –
 - number of R1 residual equations,
 - i.e., the number of R1 constraint equations)
3. When using a model with over-determined connectors, i.e., making an instance, all missing equations of this component must be provided to make the component “balanced”. Besides the standard rules, the only way to make over-determined connectors balanced is to connect to these connectors or by leaving them unconnected.

These rules shall be demonstrated at hand of the Modelica.Mechanics.MultiBody library:

The over-determined connector “Orientation” describes the transformation matrix from one frame to another frame:

```
record Orientation
  Real T[3, 3] "Transformation matrix";

  encapsulated function equalityConstraint
    import M=Modelica.Mechanics.MultiBody;
    input M.Frames.Orientation R1
    input M.Frames.Orientation R2
    output Real residue[3]
  algorithm
    residue := { ... }
  end equalityConstraint ;
end Orientation;
```

The Orientation object has a residue function with 3 equations and is used in a MultiBody connector to

describe the rotation from the world frame to the connector frame:

```
connector Frame
  import SI = Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  SI.Position r_0[3] "Origin of frame"
  flow SI.Force f [3] "Cut-forces"

  M.Frames.Orientation R "Orientation"
  flow SI.Torque t[3] "Cut-torque";
end Frame;
```

Connector frame has $3+3 = 6$ flow variables (f, t), 3 normal, non-flow variables (r_0) and 3 residual equations (from R). Therefore, the connector fulfills rule 1 above.

FixedTranslational is a MultiBody model that translates one frame along a given position vector:

```
model FixedTranslation
  import SI=Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_a frame_a
  M.Interfaces.Frame_b frame_b
  parameter SI.Position r[3]
    "Vector from frame_a to frame_b";
equation
  Connections.branch
    (frame_a.R, frame_b.R);
  frame_b.r_0 = frame_a.r_0 +
    M.Frames.resolve1(frame_a.R, r);
  frame_b.R = frame_a.R;
  zeros(3) = frame_a.f + frame_b.f;
  zeros(3) = frame_a.t + frame_b.t +
    cross(r, frame_b.f);
end FixedTranslation;
```

The number of equations in FixedTranslation is required to be:

$$\begin{aligned}
 &= 2*(3+3+9+3) // 2*(r_0+f+R.T+t) \\
 &- 2*(3+3) // 2*(f+t) \\
 &- (9-3) // (R.T - R.residuals) \\
 &= 18 \text{ equations}
 \end{aligned}$$

and the model fulfils this requirement.

World is the MultiBody model that defines the inertial frame as:

```
model World
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_b frame_b;
equation
  Connections.root(frame_b.R);
  frame_b.r_0 = zeros(3);
  frame_b.R = M.Frames.nullRotation();
end World
```

The number of equations in World is required to be:

$$\begin{aligned}
 &= 3+3+9+3 // r_0+f+R.T+t \\
 &- (3+3) // (f+t) \\
 &= 12 \text{ equations}
 \end{aligned}$$

and the model fulfils this requirement.

LineForce is a MultiBody model that defines a force along a line between two frames. The difficulty is that if LineForce elements are directly coupled to

each other, then the transformation matrix between two LineForce elements is arbitrary. This can be made mathematically well-defined, by setting one of the LineForce transformation matrices (= the selected root) to an arbitrary value:

```
model LineForce
  import SI=Modelica.SIunits;
  import M=Modelica.Mechanics.MultiBody;
  M.Interfaces.Frame_a frame_a
  M.Interfaces.Frame_b frame_b
  ...
equation
  Connections.potentialRoot(frame_a.R,10);
  Connections.potentialRoot(frame_b.R,10);

  frame_b.f = ...; // force law
  0 = frame_a.f + frame_b.f;

  if isRoot(frame_a.R) then
    frame_a.R = Frames.nullRotation();
  else
    frame_a.t = zeros(3);
  end if;

  if isRoot(frame_b.R) then
    frame_b.R = Frames.nullRotation();
  else
    frame_b.t = zeros(3);
  end if;
end LineForce;
```

The number of equations in LineForce depends on the selected roots. If `isRoot(.)` is `false` for both frames the number of equations are required to be:

$$\begin{aligned}
 &= 2*(3+3+9+3) // 2*(r_0+f+R.T+t) \\
 &- 2*(3+3) // 2*(f+t) \\
 &- 2*(9-3) // 2*(R.T - R.residuals) \\
 &= 12 \text{ equations}
 \end{aligned}$$

and the model fulfils this requirement.

If `isRoot(.)` is `false` for one and `true` for the other frame, the number of equations are required to be:

$$\begin{aligned}
 &= 2*(3+3+9+3) // 2*(r_0+f+R.T+t) \\
 &- 2*(3+3) // 2*(f+t) \\
 &- 1*(9-3) // 1*(R.T - R.residuals) \\
 &= 18 \text{ equations}
 \end{aligned}$$

and the model fulfils this requirement.

Session 1b

Language, Tools and Algorithms



Initialization of Modelica models in Scicos

Masoud Najafi Ramine Nikoukhah
INRIA-Rocquencourt, Domaine de Voluceau,
78153, Le Chesnay Cedex, France

Abstract

The current Scicos Modelica compiler only supports one form of initialization of index-1 DAEs: differential state variables and system parameters are considered known, algebraic variables as well as the derivatives of differential variables are computed. In many practical applications this is not enough. For example, often it is required to start a simulation in an equilibrium state. Scicos has recently been extended to support general initialization of Modelica models. In this paper, we present this extension and in particular the way the continuous-time part of a Modelica program is initialized, the graphical interface, as well as the initialization methods used.

Keywords: hybrid differential equations; initialization; numerical solver; Modelica; Scicos

1 Introduction

The Modelica compiler used in Scicos¹ [1, 2] has been developed in the SIMPA² project with the participation of INRIA, IMAGINE, EDF, IFP, and Cril Technology. Recently the ANR³/RNTL SIMPA2 project has been launched to develop a more complete Modelica compiler. The main objectives of this project are to extend the SIMPA compiler to fully support inheritance and hybrid systems, to give the possibility to solve inverse problems by model inversion for static and dynamic systems, and to enhance initialization of Modelica models.

A model can be simulated only if it is initialized correctly. The reason lies in the fact that a DAE (Differential-Algebraic Equation) resulting from a Modelica program can be simulated only if the initial value of all variables as well as their derivatives are known and consistent. An index-1 DAE can be formu-

lated as

$$0 = F(x', x, y, p)$$

where x, x', y, p are the vector of differential variables, derivative of differential variables, algebraic variables, and model parameters, respectively [3, 4].

Previous version of the Modelica compiler of Scicos supports only one form of initialization; it assumes that the parameters and the initial value of differential variables (variables whose derivative exist in the Modelica program) are known and given by `start` keyword, then a solver is used to compute the derivative of differential variables and algebraic variables. The user can also give start values of algebraic variables in the Modelica program which are used as guess values to help the solver to find consistent initial values. In many practical applications, this approach of initialization does not cover all forms of initializations. For example, often it is required to start a simulation in the steady state. In this case, derivatives of differential variables are set to zero and the initial values of all the variables are found as a function of known outputs (or observable states). The initialization problem can also be formulated as an inverse problem where the system outputs are known at initial time, and the inputs and internal states are to be computed. Sizing is another form of initialisation, where, *e.g.*, a parameter of the model is computed at steady states as a function of a system output (or observable states).

The next version of the Modelica compiler used in the Scicos simulator provides support for a more general form of initialization. In this initialization methodology, the user can select freely the knowns and unknowns of the initialization problem between the all Modelica variables, the derivatives of differential variables, and the parameters of the model. A confidence factor is associated to each value specifying the degree of confidence in that value. The confidence factor goes from zero to one where zero means the value is just a pure guess and one means the guess value corresponds to the actual value. This latter situation corresponds in particular to the usage of the "fixed" keyword in Mod-

¹www.scicos.org

²Simulation pour le Procédé et l'Automatique

³French National Research Agency

elica. With this initialization methodology, no matter if an unknown is a parameter or a differential or algebraic variable, it can be computed as a function of other known values.

Another problem with the previous approach of initialization concerns with start values of algebraic variables. The Modelica compiler of Scicos performs formal simplifications on Modelica programs and generates a DAE which is used for both model initialization and dynamic simulation. Using the simplified Modelica model causes a problem in initialization of big Modelica programs where guess values of algebraic variables are important for convergence of the solver. The generated DAE contains all differential variables and their derivatives, but many algebraic variables are eliminated during the formal simplification phase of the Modelica model. This has several consequences: First, algebraic variables to be eliminated may change when the Modelica model changes a bit. Thus, the user is unable to know in advance what algebraic variables will be selected to provide their guess values. Furthermore, in many cases the remained algebraic variables may not have physical interpretation and the user cannot provide the guess value without performing a calculation.

Because of these difficulties, we decided to separate the model initialization and the dynamic simulation and generate two DAEs for each one:

- a flat Modelica model without formal simplification to be used in the initialization phase
- a simplified Modelica model to be used in the dynamic simulation phase.

With this approach, the problem of the selection of algebraic variables no longer exists. In fact, once the initialization phase finished, initial values of all algebraic variables as well those of derivatives of differential variables are known. As a consequence, the selection of algebraic variables in the simplified Modelica code does not affect the dynamic simulation.

In Modelica, the `start` keyword can be used to set start values of variables. Start value of derivatives of variables can be given within `initial equation` section. For small programs, this method can be easily used but as the program size grows, it becomes difficult to set start values and change the `fixed` attribute of variables/parameters directly in the Modelica program; initialization via modifying the Modelica model is specially difficult for models with multiple level of inheritance. *e.g.*, The GUI (Graphical User

Interface) of Dymola⁴ is not well adapted to visualize and fix/relax the variables and the parameters. Then, the user often needs to have a single model but with several initialization scenarios. So, for each scenario a copy of the model should be saved. Furthermore, it is not possible to use variable attributes used in a model in another model.

Having confronted these inconveniences, we found it easier and more intuitive if start values and other attributes of variables/parameters are provided via a GUI. In the GUI, the user can easily change attributes such as `start`, `fixed`, `max`, `min`, `nominal` for variables/parameter of a model. Furthermore, it is possible to indicate whether a variable, the derivative of a variable or a parameter must be fixed or relaxed during initialization. It is also possible to save the resulting configuration in a file and use it later.

In the following sections the initialization methodology for Modelica models, the initialization GUI, and available initialization computing methods for continuous-time Modelica models (index-1 DAE) will be explained.

2 Initialization methodology for Modelica models

The first objective of the initialization is to compute consistent initial values for all variables and derivatives of differential variables (for index-1 DAEs). In order to have access to all variables, the model should be flattened, *i.e.*, a model without inheritance should be constructed. Then, in order to initialize the model, which is often a dynamic model, *i.e.*, it contains derivatives, it should be converted into a pure algebraic model. For that, all derivatives should be replaced by algebraic variables.

The procedure of initialization is given in Figure 1. The initialization is composed of:

- converting the Modelica program into a flat Modelica model
- converting the flat model to an XML file
- modifying the XML file by the initialization GUI
- converting the XML file back to a Modelica program
- computing the unknown variables/parameters.

⁴www.dymola.com

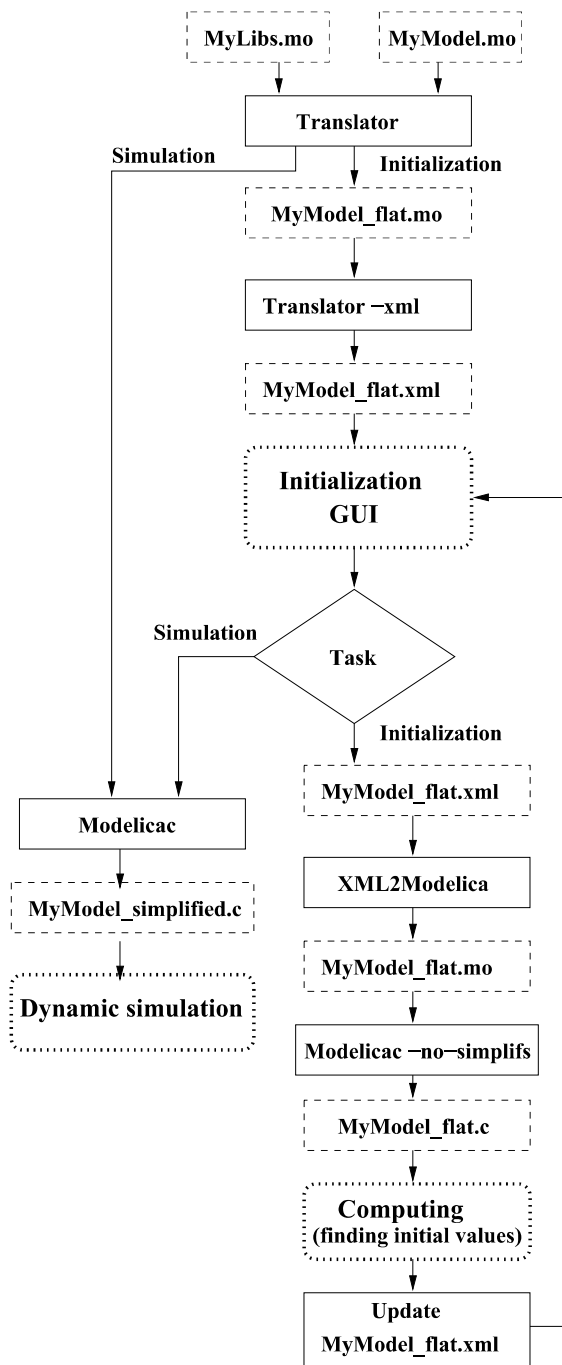


Figure 1: Initialization in Scicos

In the initialization process, three external applications are used: `Translator`, `XML2Modelica`, and `modelicac` (all developed at *LMS Imagine.Lab*⁵).

`Translator` is used for three purposes:

- Modelica Front-end for dynamic simulation. When called with appropriate options, `Translator` generates a flat Modelica program. For that, `Translator` verifies the syntax and

semantics of the Modelica program, applies inheritance rules, generates equations for connect expressions, expands for loops, handles pre-defined functions and operators, performs the implicit type conversion, and etc. The generated flat model contains all the variables, derivatives of differential variables, and parameters defined and declared with `fixed=false`. Constants and parameters with the attribute `fixed=true` are replaced by their numerical values.

- Modelica Front-end for initialization. In this case, besides generating a flat Modelica model, derivatives of the variables are replaced by an algebraic variable. As a convention, e.g., "`der(x)`" is replaced by "`__der_x`" in the generated flat model for initialization.
- XML generator. When called with `-xml` option, `Translator` generates an XML file from a flat Modelica model. The generated XML file contains all the information of the flat model. An example of an XML file is given in Appendix A.

Once the XML file generated, the user can change variable/parameter attributes. The modified XML file should be reconverted into a Modelica program to be compiled and initialized. `XML2Modelica` is used to perform this task.

`Modelicac`, which is a compiler for the subset of the Modelica language, compiles a flat Modelica model and generates a C program for Scicos targets. The main features of the compiler are the simplification of the Modelica models and the generation of the C program ready for simulation. It supports discontinuous model switching and provides the analytical Jacobian of the model. It does not support higher index DAEs. This compiler is used in initialization and dynamic simulation stages:

- when called with `-no-simplifs` option, `modelicac` generates a C program without doing formal simplification and variable elimination. Only parameters with the attribute `fixed=true` are eliminated and replaced by their numerical values.
- when called with appropriate options, `modelicac` can perform formal simplification and generate a simplified C program for dynamic simulation.

When the user requests a Modelica initialization in Scicos, as shown in Figure 1, `Translator` is called

⁵<http://www.lmsintl.com>

and first a flat Modelica model, then an XML file are generated. The XML file can then be used in the initialization GUI. The user can change the variable/parameter attributes in the XML file. The modified XML file is then translated back to a Modelica program. The Modelica program is compiled with `modelicac` and a C program is generated. The C program is used by Scicos to compute the initial value of variables/parameters. Once the initialization finished, whether succeeded or failed, the XML file is updated with the most recent results which the user can visualize and decide if the dynamic simulation should start or not.

At the beginning of the dynamic simulation, the initial values of the variables are read from the XML file and the simulation can start. The results of the dynamic simulation can also be saved in an another XML file to be used as a starting point for another simulation.

3 Initialization GUI

In order to manipulate an XML file, a GUI has been developed for Scicos. This API has been developed in TCL/TK⁶. A screenshot of this GUI is shown in Figure 2.

With this GUI, the user can Open/Merge/Close/Save XML files and visualize the attributes of variables/parameters. Each variable/parameter has several attributes: `name`, `id`, `type`, `fixed`, `value`, `weight`, `max`, `min`, `nominal`, `comment`, and `selection`.

- **name**: attribute is the name of the variable/parameter used in the Modelica program. Note that the derivative of a variable is replaced by an algebraic variable. The user cannot change the `name` attribute.
- **id**: the identification of `name` and is used to locate the variable in the XML file. The user cannot change the `id` attribute.
- **type**: it indicates whether `name` is a parameter or a variable in the original Modelica program. The user cannot change the `type` attribute.
- **fixed**: it shows the value of the `fixed` attribute of `name` in the original Modelica program. The user cannot change the `fixed` attribute.
- **value**: it is the default value of `name` in the original Modelica program. The user can modify this field.

- **weight**: it is the confidence factor attributed to `name` and can take a value in the range $[0, 1]$. `weight==0` means the value is just a pure guess. This situation corresponds to the `fixed=false` in Modelica. `weight==1` means the given value corresponds to the initial value. This situation corresponds to `fixed=true` in Modelica. The default value of `weight` for parameters and differential variables is one, whereas for algebraic variables and derivative of differential variables (converted to variables) is zero. Note that when the user sets `weight` to one, the corresponding variable/parameter will be considered as a constant and in the initialization phase, it will be replaced by its numerical value. In this way, a parameter in Modelica program can be considered as an unknown and its value be computed during the initialization phase.

For large models, it is nearly impossible to give all guess values, so the guess values of unspecified variables are set to zero by the compiler. Furthermore, many variables are redundant and the user does not know the initial guess of which ones should be given. This often happens with variables linked by the `connect` operator. Consider, *e.g.*, the following equation set.

$$F(X) : \begin{cases} 0 &= \frac{x-3}{(x-3)^2+1} - 0.1 \\ 0 &= x-y \end{cases}$$

In this case, if the user sets the initial guess of y to 10 and leaves the guess value of x unspecified *i.e.*, $x = 0$, although $y = 10$ is close to the solution, the Newton's methods will likely fail. Because, the solver ignores the initial value of y and uses that of x . In fact, there is no way to indicate the solver the guess value which is "more" correct than the others. Using the `weight` attribute can be useful to help the solver to give more importance to the values specified by the user and help the solver to converge toward the desired solution.

When the final model is not square or `weight` values are zero or one, an optimization problem should be solved to minimize the cost:

$$\text{Cost} = \lambda F(X)F^t(X) + \sum_{i=1}^N w_i(x_i - v_i)^2 \quad (1)$$

where x_i is an unknown, v_i is its guess value, and w_i is `weight` or the confidence factor. λ is the Lagrange multiplier and N is the total number of unknowns.

⁶<http://www.tcl.tk/software/tcltk/>

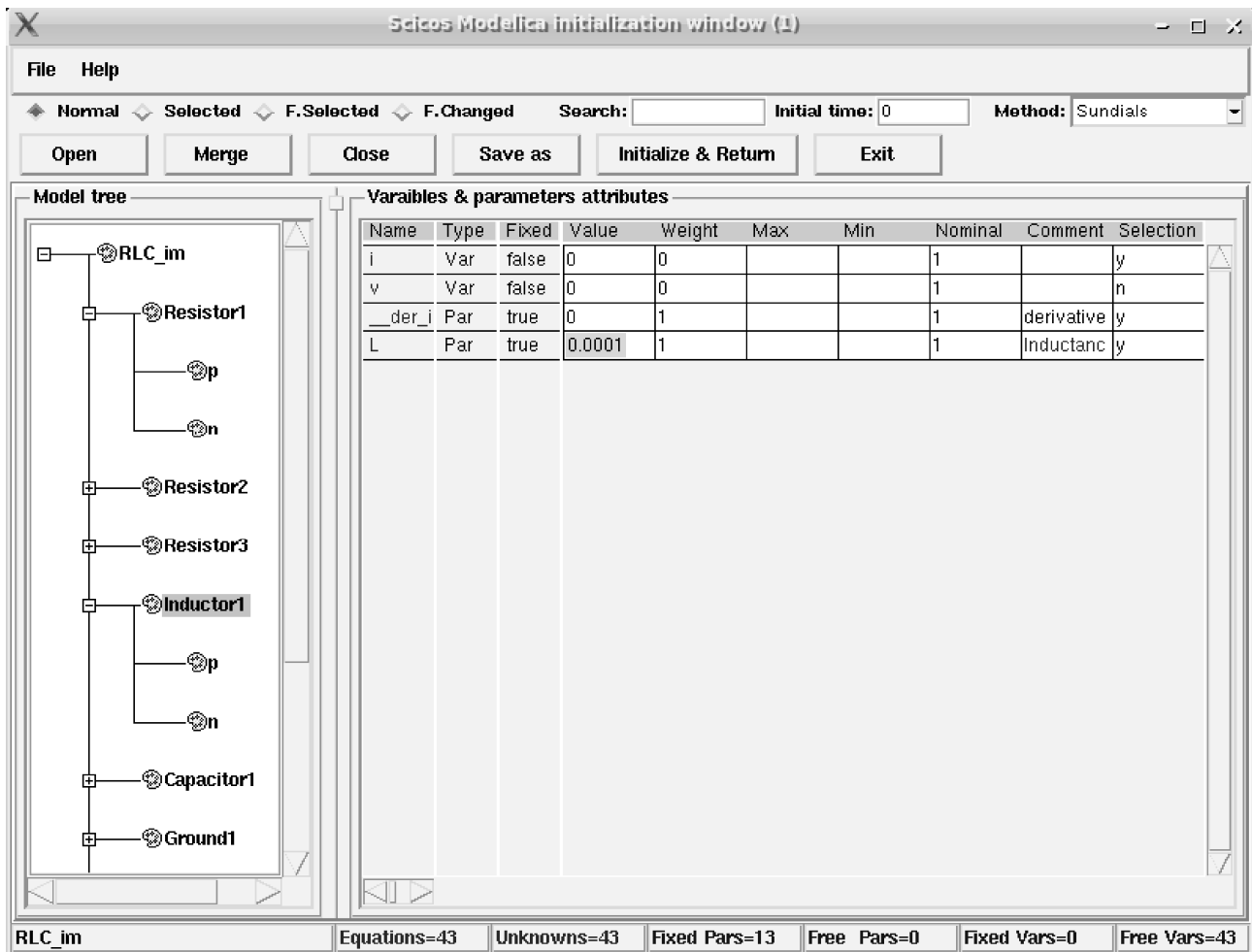


Figure 2: Screenshot of the initialization GUI in Scicos for the electrical circuit of Figure 4

- **max/min:** they are used to set maximum and minimum bounds on values.
- **nominal:** it is used to set error tolerances and normalize variables/parameters.
- **Comment:** it is the comment provided in the Modelica program for variables or parameters and can be modified by the user.
- **selection:** it is used to select and display interesting variables/parameters. If the display mode is **Selected**, only variables/parameters whose selection field is 'y' will be displayed. This option is useful specially when a block has many variables but the user is interested only in a few. There are other display modes, *i.e.*, **F.selected** which displays all selected variables, **Changed** which displays only the variables or parameters whose **weight** attributes have changed.

Once the user changes the new attributes of vari-

ables/parameters, the initialization process can be invoked by clicking on the "Initialize & Return" button. The obtained results, either successful or failed, are put back into the XML file and new values are displayed. If failed, the user can select another computing methods available and iterate until a consistent result is obtained.

4 Computation methods

The initialization problem is generally defined as the solution of a nonlinear system of equations

$$0 = F(X)$$

where X is a vector composed of all the variables, derivatives of the differential variables (transformed into algebraic variables), and relaxed parameters (**fixed=false** or **weight=0**). The initialization GUI provides several computation methods for initialization. None are guaranteed to converge, so the user

should try sequentially several available methods until one works. This is possible because, if one method fails, another method can be tried using the last obtained result. In this section these methods as well as their characteristics will be presented.

4.1 Sundials

SUNDIALS⁷ [5, 6] is a family of solvers which includes CVODE, for systems of ordinary differential equations, CVODES, variant of CVODE for sensitivity analysis, KINSOL, for systems of nonlinear algebraic equations, and IDA, for systems of differential-algebraic equations.

CVODE and IDA solvers have already been integrated in Scicos for dynamic simulation of models. IDA includes an optional user-callable module to recompute the initial conditions so as to be consistent with the given DAE system. This module uses inexact Newton methods with line-search strategies for accelerated convergence. We used this feature of IDA for initialization of models. IDA also permits inequality constraints to be imposed on the solution components. This can be used to implement max/min attributes used in Modelica.

4.2 Fsolve

Similar to IDA, Fsolve finds a zero of a system of nonlinear functions. This solver is based on the Modified Newton method and uses the QR factorization technique.

4.3 Optim

When the Modelica model is flattened, several variables exist for which start values are not given by the user. The compiler sets their start values to zero. It happens very often that the Jacobian matrix associated with the nonlinear equations in the very first step becomes singular, *e.g.*, consider the equation set (2) with $X_0 = [0, 0]^T$ start values.

$$F(X) : \begin{cases} 0 & = & x - 1 \\ 0 & = & xy - 1 \end{cases} \quad (2)$$

The first computed Jacobian matrix is singular and the Newton method fails. In order to overcome this problem, the user can choose the Optim solver. In this case, Scicos tried to obtain the solution by minimizing the cost function $C(X)$.

$$C(X) = \sum_{i=1}^N f_i^2(X) \quad (3)$$

The result of this optimization provides non-zero values that may be used as new start values for other methods, such as Sundials.

4.4 Homotopy

If Sundials or Fsolve methods fail, the user can use another methods often referred to as continuation or homotopy methods [7, 8]. Continuation methods are slower but more robust than Newton's method. In electrical circuit simulator, such as Spice and Spectre, in the case the Newton iteration fails, homotopy methods are used [9, 10, 11]. They start by modifying the model in such a way that the solution to the modified model is known or easy to compute, and such that a parameter controls the amount of the modification. Once the solution has been found for the modified model, the parameter is slowly returned to the original value, which causes the model set to return to its original form. As the parameter is changed, the solution is computed at each step, using the solution obtained from the previous step as the starting point. As long as the solution changes continuously as a function of the parameter, and the steps are small enough, the previous solution is very often a good starting point and the Newton's method converges. In other words, the model is written in the form $\phi(X, s) = 0$ where s is a real valued variable changing in the interval $[0, 1]$. In Scicos, we use

$$\phi(X, s) = s F(X) + (1 - s) (F(X) - F(X_0))$$

where X_0 is the initial starting point and $F(X)$ is the initialization equation to be solved.⁸

- for $s = 0$ the solution X_0 is known in advance or is easy to compute, *i.e.*, $\phi(X, s) = F(X) - F(X_0)$.
- for $s = 1$, the equation to be solved is the original initialization equation, *i.e.*, $\phi(X, s) = F(X)$
- the trajectory $X(s)$ is a continuous function of s .

This results in a contour curve between the solution of the initial system and the desired solution of the final system, as shown in Figure 3-A. The procedure for doing that is to slowly vary s from 0 to 1, computing the $X(s)$ at every step. In each step a the Newton method

⁸ $\phi(X, s) = s F(X) + (1 - s) (X - X_0)$ is another popular form used in homotopy methods, but we could not get good results with.

⁷<https://computation.llnl.gov/casc/sundials/>

is used to find the solution. The homotopy method is considerably slower than Newton's based method (Sundials & Fsolve).

The homotopy method may fail with some types of curves: simple discontinuities (see Figure 3-B), folds (see Figure 3-C), bifurcation (see Figure 3-D), and non-convergent curves (see Figure 3-E,F). Simple discontinuities are caused by discontinuities in the model equations. Folds, which are relatively common, occur when the solution curve doubles back on itself and results in the model having multiple solutions for at least some values of $X(s)$.

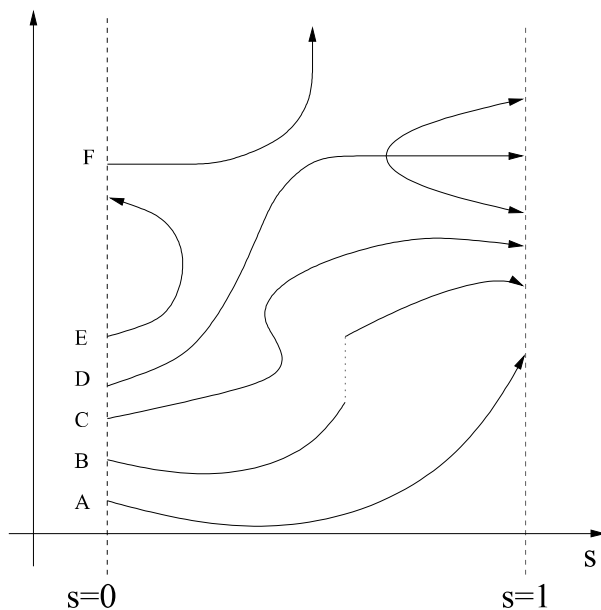


Figure 3: Homotopy curves

In the Spice and Spectre simulators, several variants of homotopy methods such as Gmin stepping and source stepping are used. In Gmin stepping, a resistor is placed between each node and ground. The resistor values are varied from zero to a very high value. In the source stepping method, the circuit sources are varied from zero to nominal values. In practice, source stepping trajectories are plagued by folds and so do not work very well. Gmin stepping is also subject to folds, but is much less susceptible to them than source stepping [9]. The Gmin stepping cannot be implemented without modifying the Modelica model, the source stepping method can, however, be employed. There are several homotopy methods and software available. In Scicos, we use HOMPACK⁹ package. This package provides several homotopy methods such as *fixed point*, *zero finding*, and *general homotopy curve*

⁹<http://www.netlib.org/hompack/>

tracking using three different algorithms: ODE-based, normal flow, and augmented Jacobian [12, 13, 14]. We got better results with the *general homotopy curve tracking* method. HOMPACK can be successfully used for curves with folds, it fails however for discontinuous curves and curves with bifurcations.

4.5 Fsolve stepping

In this method, which is also an homotopy method, the s value is increased monotonically from 0 to 1 and in each step the nonlinear function $\phi(X,s)$ is solved using Fsolve. An interesting property of Fsolve is returning the absolute value of the solution when there are two complex conjugate solutions. For example, (4) has positive real valued solutions, but the solution curve $X(s)$ from $s = 0$ to $s = 1$ is not entirely in real domain, the solutions go to the complex domain and then returns.

$$F(X) : \begin{cases} 0 & = x + y - 5 \\ 0 & = xy + y - 8 \end{cases} \quad (4)$$

The Fsolve stepping option does not support folds in the curve.

5 Example

Consider the electrical circuit shown in Figure 4. This circuit has been modeled with Modelica blocks in Scicos. The initialization GUI for this circuit is shown in Figure 2. For this electrical circuit, we would like to use the initialization GUI for two purposes: initialization from equilibrium state and for parameter sizing.

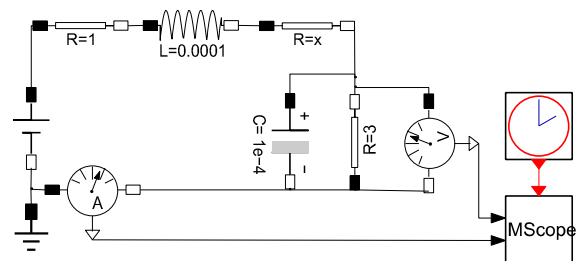


Figure 4: An electrical circuit to be initialized

- **Initialization:** In order to initialize from the steady state, all derivatives values should be fixed

to zero. For that, in the initialization GUI, derivatives of the differential variables which have been transformed into algebraic variables should be set to zero and their weight attribute should be set to one. In this circuit there are only two derivatives variables defined in inductor and capacitor blocks. In the initialization GUI, clicking on the name of these blocks, the user can change the attributes of derivatives. Then, the user can select a compute method and launch the initialization. Once the initialization finished, the obtained results will be displayed in the GUI and the user can start the dynamic simulation or retry another initialization method.

- **Parameter sizing:** In this case, the user needs to find a parameter value as a function of known system outputs. Suppose that the user needs to compute the resistance value of `Resistance1` that results in a current equal to two Amperes through the inductor at steady state. In this case, the `R` parameter in `Resistance1` should be relaxed by setting its weight attribute to zero. Then, the current through `Inductor1` should be fixed to two Amperes (by setting the weight attribute of `i` to one and its value to two). After selecting the compute method, the resistance value would be computed.

It is interesting to note that several initializations can be performed without modifying the Modelica program. Each initialization can be saved in an XML file and be used later.

6 Future works

The model initialization and model inversion which has been implemented asks some features:

- **weight:** At the current stage of the project, we focus on handling the cases `weight=0` and `weight=1` corresponding to `fixed=false` and `fixed=true`. At the next stage of the project, the user would be able to give the weight values between zero and one. In order to solve this optimization problem, we intend to implement several optimization methods such as least-square, quasi-Newton, conjugate-gradient, and Nelder-Mead (moving simplex) methods in Scicos.
- **initial equations:** They are not yet supported with `Translator`. It should be included in `Translator` by the end of the SIMPA2 project.
- **Arrays:** The way arrays and matrices are displayed in the initialization GUI should be improved. Currently, arrays are displayed element by element. When the number of elements of the array is large, this becomes inconvenient.
- **max, min, nominal:** Although implemented in the initialization GUI, they are not yet used in computing the initial values.
- **structural analysis:** In sizing or dynamic model inversion, when a fixed variable/parameter is relaxed, the user should fix a relaxed variable/parameter. The choice of variable/parameter is not straightforward and easy. In order to have a reasonable choice, a structural analysis should be performed. This task is a part of SIMPA2 project that should be implemented by 2009.
- Piecewise linear methods and Simplex are other methods that are used when the Newton based methods fail [15, 16, 17, 18, 19]. We intend to implement these methods in Scicos for model initialization.

7 Conclusion

In this paper the new initialization methodology of Scicos for initializing Modelica models has been presented. This methodology provides an easy and intuitive way for initializing Modelica models as well as model sizing and inverse problems.

References

- [1] S. L. Campbell, J.P. Chancelier, R. Nikoukhah, Modeling and simulation in Scilab/Scicos, Springer Verlag publishing, 2005.
- [2] J. P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, S. Steer, An introduction to Scilab, Springer Verlag, Le Chesnay, France, 2002.
- [3] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM publication, Philadelphia, 1996.
- [4] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, "Consistent initial condition calculation for differential-algebraic systems", *SIAM J. Sci. Comp.*, no. 19, 1998.

- [5] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," *ACM Transactions on Mathematical Software*, 31(3), pp. 363-396, 2005.
- [6] A. C. Hindmarsh, "The PVODE and IDA Algorithms," LLNL technical report UCRL-ID-141558, December 2000.
- [7] E.L. Allgower and K. Georg, *Numerical Continuation Methods, an Introduction*, Springer Ser. in Comput. Math. Springer-Verlag, Vol 13, 1990.
- [8] E.L. Allgower and K.Georg, "Continuation and path following", *Acta Numerica*, 1993, pp. 1–64.
- [9] Kenneth S. Kundert. "The designer's guide to Spice and Spectre". Kluwer academic publishers, 1995.
- [10] A. Dyess, E. Chan, H. Hofmann, W. Horia, and Lj. Trajkovic, "Simple implementations of homotopy algorithms for finding dc solutions of nonlinear circuits", *Proc. IEEE Int. Symp. Circuits and Systems*, Orlando, FL, Vol. VI, 1999, pp. 290–293.
- [11] W. Mathis, Lj. Trajkovic, M. Koch, and U. Feldmann, "Parameter embedding methods for finding dc operating points of transistor circuits", *Proc. NDES '95*, Dublin, Ireland, 1995, pp. 147–150.
- [12] L.T. Watson, *Appl. Math. Comput.* 5 (1979), pp. 297-311.
- [13] L.T. Watson, "Globally convergent homotopy algorithm for nonlinear systems of equations", *Nonlinear Dynamics*, Vol. 1, 1990, pp. 143–191.
- [14] L. T. Watson, M. Sosonkina, R. C. Melville, A. P. Morgan, and H. F. Walker, "Algorithm 777: HOMPACT90: A suite of FORTRAN 90 codes for globally convergent homotopy algorithms", *ACM Trans. Math. Software*, Vol. 23, 1997, pp. 514–549.
- [15] B. C. Eaves, "A Short Course in Solving Equations with PL Homotopies", *SIAM-AMS Proceedings IX*, 1976, pp. 73–143.
- [16] K. Georg, "An introduction to PL algorithms: Computational solution of nonlinear systems of equations", *Lectures in Applied Mathematics, American Mathematical Society*, 26, 1990, pp. 207–236.
- [17] W.P.M.H. Heemels, J.M. Schumacher, and S. Weiland, "Linear complementarity systems", *SIAM Journal on Applied Mathematics*, Vol. 60 (2000), pp. 1234–1269.
- [18] C.E. Lemke, and J.T. Howson, "Equilibrium points of bimatrix games", *SIAM J. Appl. Math.*, Vol. 12, 1964, pp. 413–423.
- [19] D. M.W. Leenaerts, W.M.G. van Bokhoven. *Piecewise Linear Modelling and Analysis*. Kluwer Academic Publishers, Boston, 1998.

A XML file example

RCL_im.xml

```

<model>
  <name>RCL_im</name>
  <elements>
    <terminal>
      <name>R3</name>
      <kind>parameter</kind>
      <id>R3</id>
      <fixed value="true"/>
      <initial_value value="(5)"/>
      <weight value="1"/>
      <max value=""/>
      <min value=""/>
      <nominal_value value="1"/>
      <comment value="R3"/>
      <selected value="y"/>
      <kind_orig>fixed_parameter</kind_orig>
    </terminal>
    .....
  <struct>
    <name>CurrentSensor1</name>
    <subnodes>
      <terminal>
        <name>i</name>
        <kind>variable</kind>
        <id>CurrentSensor1.i</id>
        <fixed value="false"/>
        <initial_value value="(0)"/>
        <weight value="0"/>
        <max value=""/>
        <min value=""/>
        <nominal_value value="1"/>
        <comment value=""/>
        <selected value="y"/>
        <kind_orig>variable</kind_orig>
      </terminal>
    </struct>
    <name>p</name>
    <subnodes>
      .....
    </struct>
  </elements>
  <equations>
    <equation value="0 = ('VoltageSensor1.n.i' + ...;"/>
    <equation value="0 = ('Ground1.p.i' + ...;"/>
    <equation value="0 = ('Resistor1.n.i' + ...;"/>

```

```
....  
</equations>  
<when_clauses/>  
</model>
```

Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems

Dirk Zimmer

Institute of Computational Science, ETH Zürich
CH-8092 Zürich, Switzerland
dzimmer@inf.ethz.ch

Abstract

This paper presents a derivative language of Modelica that is called Sol. It has been especially designed for the convenient expression and simulation of variable-structure systems within an object-oriented, equation-based modeling framework. Starting from a formal definition of the grammar and type-system, the paper advances to an explanation of Sol's semantics. Finally the current state of implementation and corresponding processing mechanisms are presented. **Keywords:** *language design, variable-structure systems, causalization mechanisms.*

1 Motivation

Many contemporary models contain structural changes at simulation run-time. These systems are typically denoted by the collective term: variable-structure systems. The motivations that lead to the generation of such systems are manifold. Typical cases are represented by ideal switching or breaking processes, variability in the number of entities, dynamic multi-level models or user interaction [5].

Let us focus on the modeling-paradigm that is represented by Modelica: declarative models that are based on differential algebraic equations (DAEs) with hybrid extensions. Within this paradigm, a structural change is reflected by a change in the set of variables and by a change in the set of relations (i.e., equations) between the time-dependent variables. Such replacements may lead to severe changes in the model structure. This concerns the causalization of the equation system, as well as the perturbation index of the DAE system.

Current contributions of this research domain include the development of the language MOSILAB [8] or Hydra [7]. Also some specific techniques, like inline-integrators [4] that can be included in Modelica prove to be helpful in certain situations. However, most of these approaches leave the standard domain of Modelica, since the modeling of variable-structure systems within the current Modelica framework is

very limited [10]. This is partly due to a number of technical restrictions that mostly originate from the static treatment of the DAEs. But these technical restrictions are not the only limiting factor. Another major problem is the lack of expressiveness in the Modelica-language itself.

To express structural changes, a corresponding modeling language has to meet certain requirements. For instance, it must be enabled to state relations between variables or sub-models in a conditional form, so that the structure can change depending on time and state. In addition, variables and sub-models should be dynamically declarable, so that the corresponding instances can be created, handled, and deleted at run time. Such requirements partly contradict with fundamental assumptions made in the design-process of Modelica.

Therefore we decided to develop a new language called Sol [11]. It is a language primarily conceived for research purposes that attempts to be of minimal complexity with a high degree of expressiveness. We want to explore the full power of a declarative modeling approach and how it can handle potential, future problem fields. The implementation of Sol will be a small and open project that should enable other researchers to test and validate their ideas with a moderate effort. The longer term goal of our research is to significantly extend Modelica's expressiveness and range of application. Furthermore, the Sol-project gives us a development-platform for technical solutions that concerns the handling of structurally changing equation systems. This includes solutions for dynamic recausalization or the dynamic handling of structural singularities.

Although Sol forms a language of its own, it is designed to be as close to Modelica as reasonably possible. This should drastically ease the understanding for anyone in the Modelica community. It is not our goal to immediately change the Modelica standard or to establish an alternative modeling language. Our scientific work is intended to merely offer suggestions and guidance for Modelica's future development.

Example 1: Model of a simple machine driving a fly-wheel with a fluctuating torque.

```
model SimpleMachine
```

```
    define inertia as 1.0;
```

```
interface:
```

```
    parameter Real meanTorque;
    static Real w;
```

```
implementation:
```

```
    static Real torque;
    static Real a;
    torque = inertia*z;
    w = int(x=z);
    phi = int(x=w);
    torque = (1+cos(p=phi))*meanTorque;
```

```
end SimpleMachine
```

```
...
```

```
static SimpleMachine M1{ meanT << 10};
cout << SimpleMachine.w;
```

Definition of the main model: “SimpleMachine”**(1) Header:**

- Definition of a constant

(2) Interface Section:

- Declaration of parameters
- Declaration of a public member

(3) Implementation part:

- Declaration of private members
- Stating Newton’s law of motion...
- Equation for the fluctuating torque

End of the model-definition

- An example declaration of the machine model.
- Simple Output Generation.

2 The Language: Sol

2.1 Principle Components

Essentially, Sol redefines the fundamental concepts of Modelica on a dynamic basis. Following the spirit of Modelica, it forms a language of strong declarative character and therefore completely abandons any imperative parts. Unlike many other declarative modeling languages, Sol enables the creation, exchange and destruction of components at simulation time. To this end, the modeler describes the system in a constructive way, where the structural changes are expressed by conditionalized declarations. These conditional parts can then get activated and deactivated during run-time. This constructive approach avoids an explicit description of modes and transitions and yet proves to be fairly powerful and flexible.

In contrast to Modelica, the grammar of Sol (cf. appendix) is significantly stricter. In its aim for simplicity, it prohibits any ambiguous ordering of its major sections. Also any grammar elements that one would typically denote by the term syntactic sugar are largely omitted. Whereas the strict section ordering definitely leads to a good modeling style, the lack of syntactic shorthand notations may sometimes result in clumsy formulations.

On the top-level, the Sol language features only a single language component that represents the definition of a model in a very generic way. Such one-component approaches are frequent for experimental languages (e.g. [1]), since they typically result in a uniform structure that eases further processing. In addition, they yield to a clear and simple grammar.

The example above gives a first glance at Sol and enables us to take a closer look at the structure of a model-definition. A model-definition consists of three parts, where each of them is optional:

- The **header** section is essentially composed out of further definitions. These may be constants or further models. Definitions of the header-part can be publicly accessed and belong to the model definition itself and not to one of its instances. In addition, the header enables you to state an extension of an existing definition.
- The **interface** section enables the modeler to declare the members of a model that can be publicly accessed. The members can either be basic variables or sub-models. Any of these members can be marked as a parameter that is passed at the model’s instantiation and remains constant for the object’s lifetime. Hence extra means for class-parameterization as in Modelica become obsolete.
- The **implementation** part contains then the actual relations between the variables and describes the dynamics of the system.

This general structure of a model-definition enables it to be used also for degenerated tasks like the definition of packages or connectors. A model in Sol represents a uniform approach that is similar to the class concept of Modelica. On the other hand side, the term “model” is almost overstressed and became so general that it lost some of its actual meaning. To regain expressiveness, Sol offers you different model-specifiers that enable the explicit denotation of certain sub-kinds. The usage of these specifiers involves consequently a number of restrictions. However, the syntax and semantics still remain uniform.

2.2 Object-Oriented Organization

The object-oriented and hierarchic organization of modeling code is substantially supported by two model-specifiers:

- **package:** Packages are used to collect models in a meaningful entity. A package-definition is reduced to the header part. It features neither an interface nor an implementation.
- **connector:** A connector typically collects members that are intended to be related by a connection-statement. A connector consists essentially of an interface. There’s no implementation part.

The creation of an object-oriented hierarchy is illustrated in example 2 where the machine-model is split up into its principle components: An engine, a flywheel and additionally a simple gear model. These models use a uniform connector model and are based upon partial models that have been collected in an extra template package. Example 2 makes frequent use of the keyword `extends` that demonstrates the appliance of type-generation.

Example 2: Package structure in Sol

```
package MechTemplate
```

```
  package Interfaces
```

```
    connector Flange
```

```
    interface:
```

```
      static potential Real phi;
```

```
      static flow Real t;
```

```
    end Flange;
```

```
    partial model OneFlange
```

```
    interface:
```

```
      static Flange f;
```

```
    end OneFlange;
```

```
      partial model TwoFlanges
      interface:
        static Flange f1;
        static Flange f2;
      end TwoFlanges;
    end Interfaces;
  end MechTemplate;

package Mechanics extends MechTemplate;

  model Engine1
    extends Interfaces.OneFlange;
  interface:
    parameter Real meanT;
  implementation:
    f.t = meanT;
  end Engine1;

  model Engine2
    extends Interfaces.OneFlange;
  interface:
    parameter Real meanT;
  implementation:
    static Real transm;
    transm = 1+cos(x = f.phi);
    f.t = meanT*transm;
  end Engine2;

  model FlyWheel
    extends Interfaces.OneFlange;
  interface:
    parameter Real inertia;
    static Real w;
  implementation:
    static Real z;
    f.phi = int(x=w);
    w = int(x=z);
    -f.t = z*inertia;
  end FlyWheel;

  model Gear
    extends Interfaces.TwoFlanges;
  interface:
    parameter Real ratio;
  implementation:
    ratio*f1.phi=f2.phi;
    -f1.t=ratio*f2.t;
  end Gear;
end Mechanics;
```

Sol offers three simple but effective mechanisms for type-generation. The most important of them is the type-extension better known as inheritance. Any model can extend any other model as long as there are no circular or recursive dependencies. Since packages represent models as well, inheritance can be applied to complete packages as well. The remaining two mechanisms consist in the redeclaration of members and the redefinition of models. Also these mechanisms can be applied to all feasible elements. In contrast to Modelica the redeclaration is used for type-generation only and not for class-parameterization.

Figure 1 depicts the resulting package structure of our example. The solid lines denote the memberships whereas the dotted arrows represent inheritance.

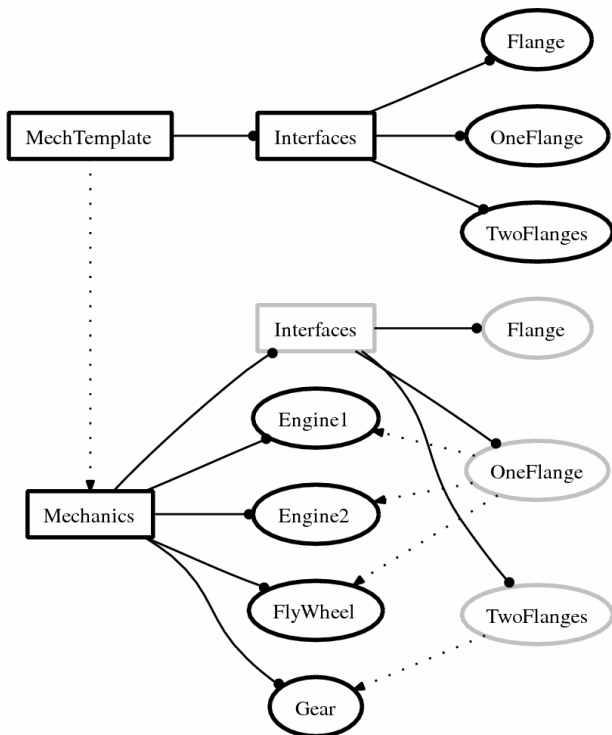


Figure 1: Exemplary package-hierarchy in Sol

Whereas the example has been over-elaborated for the purpose of demonstration, the combined usage of type-generation mechanisms forms a powerful tool for certain application domains like fluid-dynamics [3]. There, a package for a certain material may serve as a potential template. A modeler can then quickly adapt to other materials by a package-extension and a redefinition of the basic material model.

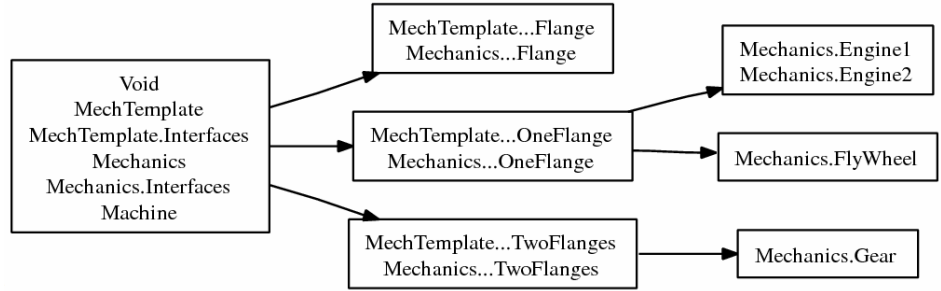


Figure 2: Exemplary type-hierarchy in Sol

2.3 Type-System

Like Modelica, Sol features a structural type-system [2]. It is solely based on the model's interfaces. The development of implementations and interfaces can therefore be separated and disjointed lines of implementation may yield into compatible types. The provided mechanisms of inheritance and redeclaration enable a satisfactory degree of polymorphism.

The type of a model is composed out of its members in the interface section. Any type-extension will yield to the creation of a sub-type of the inherited model. Also redeclarations and redefinition are limited to be only possible by sub-types of their original representation. Figure 2 illustrates the resulting type-structure of Example 2.

A proper and user-evident type-system becomes increasingly important in a dynamic framework like Sol. In situation where assignments are applied on complete sub-models to perform a model-exchange the corresponding assignments should be guarded by the type-rules.

2.4 Implementation part

The implementation part represents a block. A block may contain declarations of private members, relations (e.g. equations) or further nested conditional blocks in any arbitrary order. Let us analyze each component in more detail.

Private Declarations:

Declarations of private members do hardly differ from their counterparts in the header sections. Only the parameter attribute and the access-specifiers are now meaningless and therefore disabled.

The declaration of a member links a model instance to a given identifier. This linking is either static or dynamic. This selection has to be stated before the actual declaration. In contrast to a static linking, a dynamic linking enables to modeler to (re-)assign a new instance to the corresponding identifier.

Conditional Blocks:

Sol features if-else-branches and when-else-branches. The condition of an if-branch is immediately applied. It forms a safe condition that can be assumed to hold for its content. Hence the condition must be independent on any of its branches' content. When-statements are used to catch an event. The events are triggered during the update-procedure and are scheduled for the next one. Thus, when-conditions are not safe. Unlike Modelica, there are no syntactical restrictions on the content of the branches, but all branches shall finally lead to correct system of equations.

Statements:

Three fundamental operators are provided for setting up relations between members:

- The operator = states an equation between two expressions of type real.
- The causal copy-transmission << is setting up causal relationships between real variables and can be used to link a copy of a model-instance to an identifier.
- The causal move-transmission <- is used to link a model-instance to a new identifier and to remove the former linking.

Member-access in statements:

To access the public members of your sub-models, three options are provided:

- As in Modelica the . operator is the most straightforward way of access, but not always convenient.
- The connection(...) statement exist also in Sol and has practically the same meaning as its counterpart in Modelica.
- The () operator enables a function-like notation. It is especially suited for anonymously declared members.

Whereas the . operator represents a universal form of member access, the other two forms serve convenience and their proper appliance is determined by specifiers at the corresponding member-declarations. The connection statement only refers to variables that have been marked by the specifiers **flow** or **potential**. The specifiers **in** and **out** determine the applicability of the access by round-brackets.

3 Example Model

The presented language elements are sufficient for the formulation of highly variable systems. However,

given the brief introduction above, it may not be evident how objects can be dynamically created, exchanged and deleted as there appears to be no explicit tool for these purposes. Let us therefore look at an example.

We reassemble the machine-model from example 1 that consists of an engine that drives a fly-wheel. This time we use the components of the `Mechanics` package in example 2. Furthermore we add a simple gear to our model. We recognize that the package provides two models for an engine: The first model `Engine1` applies a constant torque on the flange. In the second model `Engine2`, the torque is dependent on the positional state, roughly emulating a piston-engine. Both models share the same type (see figure 2). Our intention is to use the latter, more detailed model at the machine's start and to switch to the simpler, former model as soon as the wheel's inertia starts to flatten out the fluctuation of the torque. This exchange of the engine-model represents a simple structural change on run-time.

Example 3: Machine with a structural change

```

model Machine
implementation:
  static Mechanics.FlyWheel F{inertia<<1};
  static Mechanics.Gear G{ratio << 1.8};
  connection(G.f2,F.f);

  static Boolean fast;
  if fast then
    static Mechanics.Engine1 E{meanT<<10};
    connection(E.f,G.f1);
  else then
    static Mechanics.Engine2 E{meanT<<10};
    connection(E.f,G.f1);
  end;

  if initial then fast << false; end;
  when F.w > 40 then fast << true; end;
end Machine;

```

The resulting model is presented above. It includes two conditional branches, one for each mode. The current mode is stored in the Boolean variable `fast`. The corresponding transition is modeled by the when-statement.

3.1 Simulation Result

Using an interpreter program, the system was simulated for 10 seconds by the excessive number of 10'000 integration steps with the forward Euler method. The computational effort sums up to a total of 0.2 seconds on a standard CPU, where the effort for parsing and preprocessing is almost completely

negligible. Figure 3 displays a plot of the angular velocity. The structural change reveals more clearly in the magnification. The actual change in the structure of equations is presented by the two causality-graphs of figure 5 and 6. Their closer examination is part of section 4.

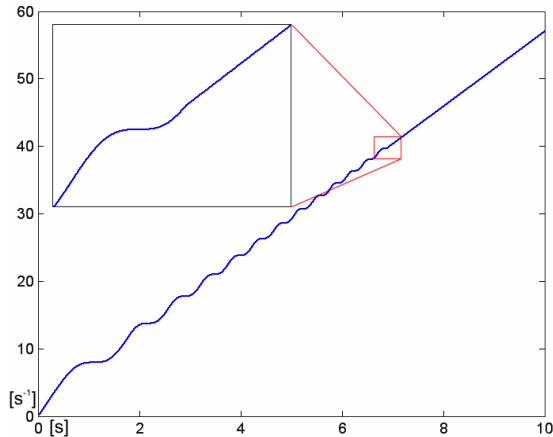


Figure 3: Angular velocity of the flywheel.

3.2 Alternative modeling approach

In the prior example, model-instances have been implicitly created and removed by the if-statement. Using local engine-models in the two branches is a very natural modeling approach, but often leads to redundant formulations (e.g. the connection statement) and therefore not all structural changes can be formulated in such a way. Thus, Sol enables the dynamic linking of an identifier to its instance. This offers a more convenient and general approach.

Let us model the machine for a second time, this time using a dynamic engine-model `E` that is initially linked to an `Engine2` model. At the transition-event, the `Engine1` model is dynamically created by an anonymous declaration. Since it is linked to the member `E` by a move-transmission, its lifetime exceeds the event and the newly created model replaces the former one. The replacement is valid because the types of the two engine models are equivalent.

Example 4: Alternative version of the machine-model

```

model Machine
implementation:
  static FlyWheel F{inertia<<1};
  static Mechanics.Gear G{ratio << 1.8};
  dynamic Engine2 E{meanT << 10};
  connection(E.f,G.f1);
  connection(G.f2,F.f);
  when F.w > 40 then
    E <- Engine1{meanT << 10};
  end;
end Machine;

```

The deletion of a model-instance is mostly done implicit by replacing the linking to an instance (as above) or by the removal of the corresponding identifier. However, example 5 presents the predefined `trash` object that is of type `void` and can be used for the explicit deletion of any object.

Example 5: Explicit deletion of a model-instance

```
trash <- E;
```

This mechanism for the dynamic linking of a model-instance represents a pointer-free modeling approach. The linking obeys clear ownership principles and therefore the simulation system can assure a memory-safe execution. Furthermore, the modeler is freed from the tedious and error-prone task of memory-management.

4 Processing Schemes

Sol is currently processed by an interpreter. The interpreter was named Solsim and represents a command-line program running under Linux or Windows. The input-file can be written in a standard text-editor. The simulation is performed and its output can be written into a file readable by the programs Matlab™ or Gnuplot. In addition to its main task, the interpreter provides also tools for the analysis of the model-hierarchy, type-structure and causalization mechanisms

Whereas the pair of a compiler and a simulator is the preferred choice for high-end simulation tasks, an interpreter is an appropriate tool (cf. [6]) for research work on language design. The development process becomes easier, faster and more flexible. Hence the development of the interpreter can proceed in parallel with a further refinement of the language. Furthermore, new debugging techniques become crucial in a more dynamic framework. This can be easier provided by an interpreter, since all necessary meta-information is available. Figure 4 displays a simplified overview of the main processing scheme that is composed out of six blocks. The following sections discuss these parts in more detail.

4.1 Parsing and Lexing

The Lexer processes the elementary elements of the language and discards all comments and formatting. Since the remaining part of the language forms an L1-Grammar, the actual parsing forms a rather trivial task. The parser is handwritten and features an automatic error-generation.

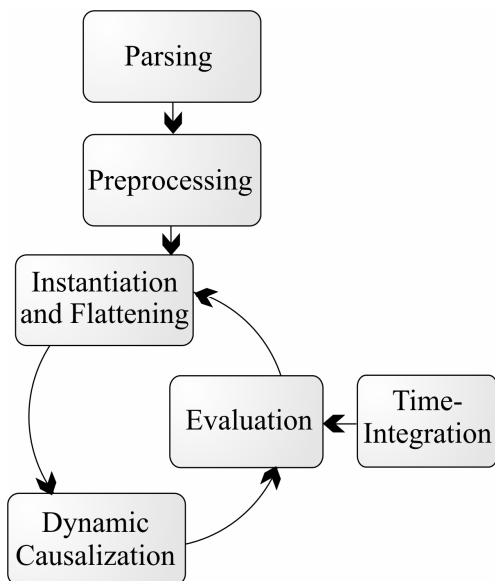


Figure 4: Processing scheme of Sol

4.2 Preprocessing

In the next stage, the mechanisms for type-generation are applied. This concerns primarily the resolving of type-identifiers and the appliance of the type-extensions. However these two processes cannot be implemented in a linear fashion. They usually have to be processed in several, interleaved steps.

Since a type-extension can be applied even on a complete package, the extension itself may generate new type-identifiers that may have to be resolved elsewhere. Thus, the algorithm has to “crawl” through the dependencies. Circular or recursive extensions lead to an inevitable downfall of this process and are therefore detected.

Furthermore the mechanisms for model-redefinition and member-redeclaration are processed. All methods for type-generation undergo a validation process, where consistency of the type-structure is checked. The resulting tree-structure of the package-hierarchy and of the type-system can be displayed by the interpreter. Please note that figure 1 and 2 represent graphs that have been automatically generated.

4.3 Instantiation and flattening

At the beginning, the top model is instantiated. The instantiation of a model evokes the following steps: First, all members (i.e. variables or sub-models) are instantiated recursively. Second all the statements in the implementation are processed.

The process of instantiation is aligned with the flattening of the system. Hence common statements like transmissions or equations are collected in a global set. The processing of an if-statements leads prelimi-

nary just to the instantiation of its corresponding condition. The actual content is instanced at a latter evaluation cycle.

In the dynamic framework of Sol the instantiation of models isn’t restricted to the initial build up phase. Later instantiations will most likely occur. Consequently also their removal has to be managed. This is done in the exact reverse way.

4.4 Dynamic Causalization

The result of the previous stage is a flattened model represented by a global set of equations and transmissions. The dynamic causalization analyzes this set of equations generates a data-structure that is suited for later evaluation cycles. The final target of this processing stage is depicted by the causality-graph in figure 5 and 6. There, the actual change in structure is revealed.

The resulting graph sketches the dependencies between the equations and transmissions. It includes also logical dependencies (dotted-lines) that result out of the conditional branches. This graph can then be further simplified by removing alias-variables or constant parts.

Any change in the set of equations will yield to an update of the causality-graph. The new equations need to be causalized and integrated into the graph. Furthermore the causality of previously causalized equations may now change. To handle all these cases in an efficient manner, the algorithm for the dynamic causalization is strongly optimistic. This means that it preserves existing structures, as long as possible, even if they temporarily loose their causal roots. Hence we can ensure that a small local change will not cause a global change unless the structure of the equation system makes this inevitable. For instance, the exchange of the engine model will not affect the causality of the fly-wheel or the gear model. Therefore the update considers only a sub-graph and can be treated locally. The details of this algorithm remain to be published.

4.5 Update and Evaluation

Based on the causality-graph, the system can be evaluated. This may consider the whole system or only a small subpart. Arbitrary updates can be triggered. If several updates are triggered at once, they are evaluated synchronously. The update procedure evaluates all dependent relations and successfully avoids any multiple evaluations of relations where separate update-paths meet.

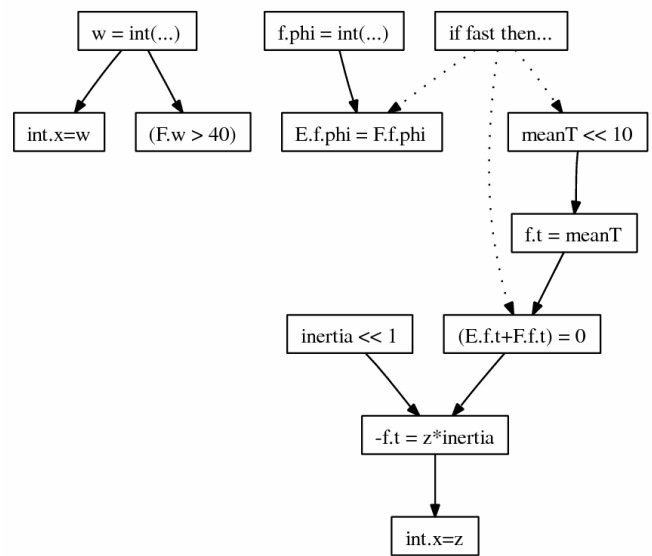
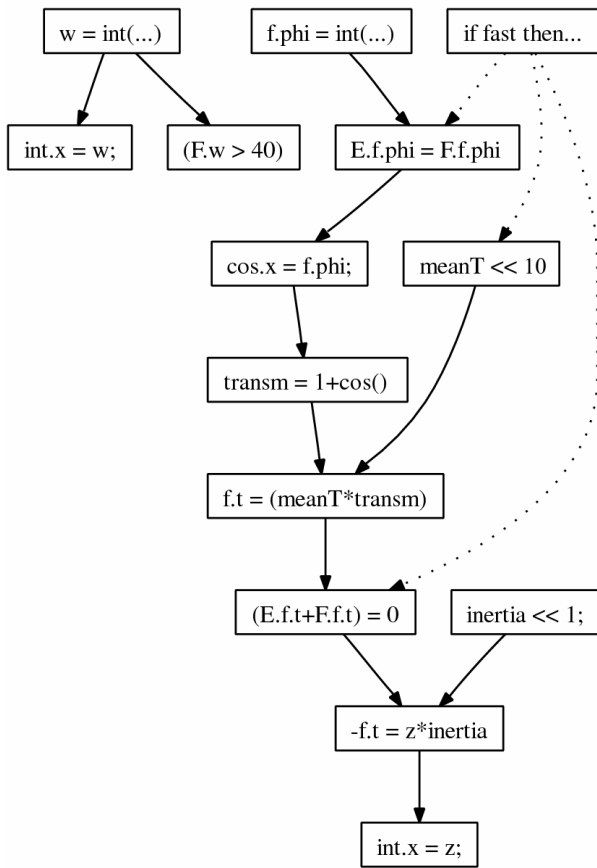


Figure 5 (left): Causality-graph of the machine at time 1 having “Engine2” as submodel.

Figure 6 (right): Causality-graph of the machine at time 8 using the simpler model “Engine1” as submodel.

Both graphs originate from an automatically generated version, where the gear-model has been omitted. The graphs have been slightly simplified to increase clarity and readability.

Logic-dependencies in the causality graph form silent dependencies. This means that an update of the corresponding Boolean expression does not directly trigger updates on its logical dependent equations. Silent dependencies are purposed only to ensure a correct update flow.

Furthermore, the causality graph contains also relations that own side-effects. Those relations may typically trigger an instantiation or removal of equations. The condition of an if-branch represents a prime example for this.

4.6 Time-Integration

The evaluation of the system (or a part of it) is triggered by two major sources. One is the insertion of new relations through instantiation. The other one, and much more frequent, is the time-integration of the corresponding state-variables. Currently, only simple explicit methods for integration are available. Since the system may reconfigure during an integration step, most integration algorithms with multiple steps cannot be implemented in a straightforward manner. It should be ensured that only the final step may trigger structural changes. Also certain methods for step-size control need to be adapted for the new framework.

5 Limitations and Efficiency

5.1 Current limitations

The current version of Solsim provides a framework for a more dynamic handling in equation-based modeling. The language itself enables the statement of drastic structural changes in a general way. Thus, the causalization of several equations may change in dependence of the structure. Also various sub-models may be instantiated or removed on run-time leading to a variable number of instances.

However, there are severe restrictions that consider the type of equation systems that are currently supported. Solsim is yet unable to treat any equation system that contains algebraic loops. Also there is no index-reduction mechanism. And therefore the differential equations are temporarily formulated by the explicit statement of an integrator.

These restrictions reduce severely the applicability of the current system. In most practical situations, structural changes hardly lead to an isolated reconfiguration like a simple causality change. Often a complete set of tasks has to be accomplished at once [5]. This concerns, for instance, the dynamic handling of algebraic loops, a dynamic state-selection

and mechanisms for index-reduction or robust, redundant re-initialization. In mechanics, the problem of multiple contact points with ideal-friction even yields to a complicated optimization task [9].

5.2 Efficiency

Whereas it is too early to give serious benchmark results, this section may at least give an impression about the current speed of our interpreter on a standard CPU. In general, we can state that the number of equations that can be evaluated per second is in the order of magnitude from 10^5 to 10^6 . The mechanisms for instantiation, flattening and causalization manage altogether to handle between 10^4 and 10^5 equations per second.

Most important is that the efficiency is high enough to let us exceed the complexity of trivial models. Of course, the interpreter, like any other interpreter suffers from a certain computational overhead that will prevent its usage for highly demanding simulation applications.

Please note that the outlined processing scheme is not an exclusive solution. It is a very general approach and consequently represents overkill for many specific applications. However, a declarative language as Sol is very well suited to enable various optimization techniques, since the semantics do not directly stipulate the processing scheme. A number of optimizations may therefore be developed. For instance, a potential optimization is a run-time compiler. One might also try to include certain parts of the causalization, simplification and flattening into the preprocessing stage. Another interesting topic is the automatic identification and pre-compilation of situations where the system can be described by a finite set of sub-modes.

5.3 Future Tasks

Our primary target is to enhance the general applicability of our approach with respect to the set of DAE-systems that can be properly handled. Therefore we have a strong incentive to develop algorithms for the tearing of algebraic loops and index-reduction that are flexible and can be well integrated into our dynamic framework.

Furthermore the presentation of the core language omits a number of language elements that have still remained in the state of design. This concerns, for example, a general solution for collections of models (e.g. arrays).

6 Conclusions

The Sol language is built upon declarative principles and is strongly influenced by Modelica. It incorporates a general modeling methodology for variable-structure systems. The Sol research project offers a dynamic framework that enables the convenient acquaintance of knowledge in language design and processing techniques that we think will be essential for Modelica's future development.

Such a methodology benefits prevalent application areas and is likely to enlarge application field for equation-based modeling. To this end, future developments that concern primarily language design and processing techniques are required.

Appendix

The following listing of rules in extended Backus-Naur form (EBNF) presents the core grammar of the Sol modeling language. The rules are listed in a top-down manner listing the high-level constructs first and breaking them down into simpler ones. Non-terminal symbols start with a capital letter and are written in bold. Terminal symbols are written in small letters. Special terminal operator signs are marked by quotes. Rules may wrap over several lines.

Common fundamental expressions like the model for the mathematical function `sin()` or given global variables as **time** or **initial** form predefined elements within the language itself and are therefore not part of the grammar. The same holds for the fundamental types in Modelica. These are: **Real**, **Integer**, **Boolean**, **String** and **Void**.

Listing 1: EBNF-Grammar of Sol

Model	=	ModelSpec Id Header
		[Interface] [Implemen] end Id ";"
ModelSpec	=	[redefine] [partial]
		(model package connector record)
Header	=	{Extension} {Define} {Model}
Extension	=	extends Designator ";"
Define	=	define (Const Designator) as Id ";"
Interface	=	interface ":" {(IDecl ParDecl) ";" } {Model}
ParDecl	=	parameter Decl
IDecl	=	[redelcare] LinkSpec [IOSpec] [CSpec] Decl
ConSpec	=	potential flow
IOSpec	=	in out
Implemen	=	implementation ":" StmtList
StmtList	=	[Statement {" ;" Statement } }

Statement	= [Condition Event Declaration Relation]
Condition	= if Expression then StmtList ElseCond
ElseCond	= (else Condition) ((else then StmtList) end [if])
Event	= when Expression then StmtList ElseEvent
ElseEvent	= (else Event)((else then StmtList) end [when])
Declaration	= [redeclare] LinkSpec Decl
LinkSpec	= static dynamic
Decl	= Designator Id [ParList]
Relation	= Expression Rhs
Rhs	= ("=" "<<" "<-") Expression
ParList	= "{" [Designator Rhs {" , " Designator Rhs }] "
InList	= "(" [Designator Rhs {" , " Designator Rhs }] "
Expression	= Comparis {(and or) Comparis }
Comparis	= Term [{"<" "<=" "==" ">" >=" >"}Term]
Term	= Product { ("+" "-") Product }
Product	= Power { ("*" "/") Power }
Power	= SElement { (" ^ ") SElement }
SElement	= ["+" "-" not] Element
Element	= Const Designator [InList] [ParList] "(" Expression ")"
Designator	= Id { ("." Id) }
Id	= Letter { Digit Letter }
Const	= Number Text true false
Number	= ["+" "-"] Digit { Digit } "." { Digit } [e ["+" "-"] Digit { Digit }]
Text	= "" { any character } ""
Letter	= "a" ... "z" "A" ... "Z" "_"
Digit	= "0" ... "9"

- [4] Cellier, F.E., Krebs, M.: Analysis and Simulation of Variable Structure Systems Using Bond Graphs and Inline Integration. In: *Proc. ICBGM'07, 8th SCS Intl. Conf. on Bond Graph Modeling and Simulation*, San Diego, CA (2007) 29-34.
- [5] Enge, O.: *Analyse und Synthese elektromechanischer Systeme*, Ph.D. Dissertation, TU Chemnitz, Germany (2006).
- [6] Mosterman, P.J.: HYBRSIM - A Modeling and Simulation Environment for Hybrid Bond Graphs, In: *J. Systems and Control Engineering*, 216, Part I (2002) 35-46.
- [7] Nilsson, H., Peterson, J., Hudak, P.: Functional Hybrid Modeling from an Object-Oriented Perspective In: *Proc. of the 1st Intern. Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany (2007) 71-87.
- [8] Nytsch-Geusen, C., et al.: Advanced modeling and simulation techniques in MOSILAB: A system development case study. In: *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria (2006) Vol. 1, 63-71.
- [9] Pfeiffer, F., Glocker, C.: *Multibody Dynamics with Unilateral Contacts*. John Wiley & Sons, New York (1996).
- [10] Zauner, G., Leitner, D., Breitenacker, F.: Modeling Structural-Dynamics Systems in Modelica [...] Mosilab and AnyLogic. In: *Proc. of the 1st Intern. Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany (2007) 71-87.
- [11] Zimmer, D.: Enhancing Modelica towards variable structure systems. In: *Proc. of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany (2007) 61-70.

Acknowledgments

I would like to thank Prof. Dr. François E. Cellier for his helpful advice and support. This research project is sponsored by the Swiss National Science Foundation (SNF Project No. 200021-117619/1).

References

- [1] Bläser, L.: A Component Language for Structured Parallel Programming. In: *Joint Modular Languages Conference*, Oxford, UK (2006) 230-250.
- [2] Broman, D., Fritzson, P., Furic, S.: Types in the Modelica Language. In: *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria (2006) Vol. 1, 303-315.
- [3] Casella, F., et al.: The Modelica Fluid and Media Library [...]. In: *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria (2006) Vol. 2, 631-640.

Biography



Dirk Zimmer received his MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He gained additional experience in Modelica and in the field of modeling mechanical systems during an internship at the German Aerospace Center DLR 2005. Dirk Zimmer is currently pursuing a PhD degree with a dissertation related to computer simulation and modeling under the guidance of Profs. François E. Cellier and Walter Gander. His current research interests focus on the simulation and modeling of physical systems with a dynamically changing structure.

Optimica—An Extension of Modelica Supporting Dynamic Optimization

Johan Åkesson
Department of Automatic Control
Faculty of Engineering
Lund University
BOX 118, SE-221 00 Lund

Abstract

In this paper, an extension of Modelica, entitled Optimica, is presented. Optimica extends Modelica with language constructs that enable formulation of dynamic optimization problems based on Modelica models. There are several important design problems that can be addressed by means of dynamic optimization, in a wide range of domains. Examples include, minimum-time problems, parameter estimation problems, and on-line optimization control strategies. The Optimica extension is supported by a prototype compiler, the Optimica compiler, which has been used successfully in case studies.

Keywords: Optimica, Language extension, Dynamic optimization, The JModelica compiler

1 Introduction

Modelica is becoming a standard format for describing and communicating high-fidelity models of large-scale dynamic systems. Expert knowledge is being encoded into Modelica libraries, both in industry and in academia. The growing body of Modelica models also represents significant capital investments, and accordingly, Modelica models and libraries represent valuable assets for many companies. As a consequence, Modelica models are turning into legacy code, which cannot easily be replaced, simply because the cost of re-encoding the models in a different format is too large.

While the primary usage of Modelica models today is simulation, several other usages are emerging. Since it is not feasible, for the reasons mentioned above, to re-encode models for each new model usage, future Modelica tools, and also the Modelica language itself, should accommodate and promote new usages

of Modelica models. This requirement has profound consequences for software design of Modelica tools, and also for the language design itself. In particular, some new usages may require new constructs, at the language level, in order to enable modeling of particular design problems.

One example of an emerging usage of Modelica models is dynamic optimization. A characteristic feature of realistic dynamic optimization problems is that the procedure of formulating such problems is highly iterative. It is common that extensive tuning of the cost function and constraints is required in order to obtain an acceptable solution. If a numerical algorithm is used to solve the dynamic optimization problem, there is an additional dimension that requires attention: the design of the transcription scheme. The scheme used to discretize the control and state variables often strongly influences the properties of the resulting solution. The choice of discretization method also affects the execution time for solving the problem, which is an important aspect in on-line applications. For these reasons, dynamic optimization problems are very rich in the sense that there are several aspects that require attention. Also, the user needs, and should be enabled to, model, using high-level language constructs, the optimization problem both in terms of cost functions and constraints and at the transcription level.

Sophisticated numerical optimization algorithms often have cumbersome APIs, which do not always match the engineering need for high-level description formats. For example, it is not uncommon for such numerical packages to be written in C, or in Fortran, and that they require the dynamic system to be modeled as an ODE/DAE, which is also encoded in C or Fortran. In addition, it may be required to also encode first and second order derivatives. Although there are efficient tools for automatic differentiation, encoding

of dynamic optimization problems in low-level languages¹ like C or FORTRAN is often cumbersome and error-prone. An important goal of developing high-level languages for dynamic optimization is therefore to bridge the gap between the engineering need for high-level descriptions and the APIs of numerical algorithms.

There are several software packages supporting dynamic optimization, for example Dymola [6] and gPROMS [12]. However, most available software tools are restricted in the sense that they usually only support a particular optimization algorithm. While a particular algorithm may work well in some cases, the appropriate choice of numerical algorithm is usually dependent on the particular problem at hand. An analogy with differential equation solvers can be made. Stiff systems call for sophisticated, but potentially computationally demanding solvers, whereas less difficult systems may be more efficiently solved by a simpler algorithm. An additional goal in the development of tools supporting high-level formulation of dynamic optimization problems is therefore to provide an open architecture, where several different algorithms can be integrated.

In this paper, an extension of Modelica, entitled Optimica, will be presented. Optimica consists of a number of new language elements, which enable high-level formulation of dynamic optimization problems based on Modelica models. The syntax as well as the semantics of Optimica will be described. In addition, a prototype implementation of an Optimica compiler, which is a modular extension of the JModelica compiler [2, 1], will be presented.

The paper is organized as follows. In Section 2, issues related to extensions of languages are discussed. Different options regarding language extensions in Modelica are also treated. In Section 3, the scope of Optimica is discussed, i.e., the class of optimization problems that can be expressed using Optimica is defined. In Section 4 the syntax and the semantics of the Optimica extension are presented. Implementation issues related to the modular Optimica extension of the JModelica compiler are discussed in Section 5. The paper ends with a summary and conclusions in Section 6.

¹The term low-level is relative, but is here used in relation to domain-specific languages like Modelica.

2 Motivation of the Optimica Extension

2.1 Isn't Modelica Enough?

Although being a very rich language in terms of expressive power for describing complex hybrid dynamical systems, Modelica lacks important features desirable for expressing optimization problems. This is quite natural, since Modelica was not developed with optimization in mind. For example, the notion of cost functions, constraints, variable bounds and initial guesses are not included in the Modelica language. Some of these quantities may indeed be modeled using standard Modelica, to some extent. For example, a particular variable may be given the meaning of cost, and the `min` and `max` attributes may be interpreted as variable bounds. However, while this approach may work in simple cases, it becomes intractable for more complex optimization problems. For example, complicated constraints, several use cases, and tailoring of the transcription method would be difficult to express. Further, the `min` and `max` attributes are usually used to express regions of validity for a model, and giving them a new semantic meaning would be potentially misleading.

2.2 What About Annotations?

Modelica offers a mechanism for adding information to model, which may not be part of the actual mathematical description, but which is convenient to store in the model. Typical examples include graphical annotations and documentation. Annotations can also be used to supply information that can be used by a particular tool, for example, in order to influence properties of the translation process. In principle, it would be possible to specify parts of an optimization problem by introducing suitable annotations. For example, a variable could be marked as a cost function, and the semantic meaning of the equality operator in an equation could be changed to that of the inequality operator. There are two reasons why it is not a good idea to strictly use this approach. Firstly, and most importantly, annotations are designed to supply *complementary* information, whereas in this case, the elements of an optimization problem are rather *primary* information, that is essential for solving the actual problem. Also, since annotations are not intended for formulation of design problems, they do not provide a convenient modeling environment for the user. Secondly, annotations cannot currently be changed by means of

modifications. Since modification is one of the cornerstones of Modelica, this is a severe restriction. Also, it is not currently well defined how annotations are treated in the case of inheritance. Since one of the main objectives of the Optimica extension is to enable convenient formulation of dynamic optimization problems using high-level constructs, using only annotations does not seem to be a feasible alternative.

Whereas the above arguments are applicable to core elements of an optimization problem, such as cost function and constraints, annotations may well be used to specify a *solution algorithm*, and associated parameters. This type of information is not part of the actual optimization formulation, but it might still be essential in order to efficiently solve the problem numerically. By introducing annotations for specifying, for example, the collocation scheme used in a direct method, the user is able to model both the actual optimization problem at hand and the transcription method in a unified high-level description language. This approach is also in line with the intentions of Modelica annotations, because of the separation between formulation of the actual problem (by means of dedicated language constructs), and specification of the solution technique (by means of annotations).

2.3 Tool-oriented Support for Optimization?

Another potential strategy for enabling dynamic optimization of Modelica models is to develop tool-oriented solutions, for example Graphical User Interfaces (GUIs), within a simulation-based software tool. This approach is used, for example, to enable optimization of Modelica models in Dymola. The user would then set up the optimization problem by entering information in dedicated fields in the GUI. Using this approach, the software tool needs to maintain an internal model of the optimization problem, as specified by the user. While this solution may be an attractive choice for interfacing a particular optimization method with existing simulation-based tools, it does not offer the flexibility, or portability, which is inherent in the Modelica language. It is therefore desirable to define, at the language level, a generic extension, which has a well defined syntax and semantics. Nevertheless, it may still be desirable to offer GUIs, in order to increase productivity in the design process, in the same way as current Modelica tools typically offer GUIs to simplify critical modeling tasks.

2.4 To Extend or to Complement?

A key issue is whether to extend Modelica by introducing new language constructs, or to define a new, separate, language which complements Modelica. By introducing a new language, the syntax and semantics of Modelica would be kept entirely intact, which may be advantageous since it makes design and maintenance of the language simpler. Also, if several extensions are introduced, defining the interaction between the extensions, both at a syntactic and semantic level, may be difficult. On the other hand, Modelica has many generic built-in constructs, e.g., classes, functions and declarative equations, which are widely applicable in many contexts. Reinventing such constructs in new languages does not seem to be an attractive alternative. Another argument in favor of language extension is that Modelica offers strong support for modularization of models. In the case of dynamic optimization, the user may construct the model separately from the formulation of the optimization problem, in which the model is used. In this way, the same model may still be used for other purposes than optimization, such as, for example, simulation.

It is essential, however, that language extensions targeted at particular usages of Modelica models do not interfere unnecessarily with the original language. Preferably, extensions should be *modular*, in the sense that the new constructs are only allowed in a well defined language environment.

3 Scope of Optimica

3.1 Information Structure

In order to formulate a dynamic optimization problem, to be solved by a numerical algorithm, the user must supply different kinds of information. It is natural to categorize this information into three levels, corresponding to increasing levels of detail.

- **Level I.** At the mathematical level, a canonical formulation of a dynamic optimization problem is given. This include variables and parameters to optimize, cost function to minimize, constraints, and the Modelica model constituting the dynamic constraint. The optimization problem formulated at this level is in general infinite dimensional, and is thereby only partial in the respect that it cannot be directly used by a numerical algorithm without additional information, for example, concerning transcription of continuous variables.

- **Level II.** At the transcription level, a method for translating the problem from an infinite dimensional problem to a finite dimensional problem needs to be provided. This might include discretization meshes as well as initial guesses for optimization parameters and variables. It should be noticed that the information required at this level is dependent on the numerical algorithm that is used to solve the problem.
- **Level III.** At the algorithm level, information such as tolerances and algorithm control parameters may be given. Such parameters are often critical in order to achieve acceptable performance in terms of convergence, numerical reliability, and speed.

An important issue to address is whether information associated with all levels should be given in the language extension. In Modelica, only information corresponding to Level I is expressed in the actual model description. Existing Modelica tools then typically use automatic algorithms for critical tasks such as state selection and calculation of consistent initial conditions, although the algorithms can be influenced by the user via the Modelica code, by means of annotations, or attributes, such as `StateSelect`. Yet other information, such as choice of solver, tolerances and simulation horizon is provided directly to the tool, either by means of a graphical user interface, a script language, or alternatively, in annotations.

For dynamic optimization, the situation is similar, but the need for user input at the algorithm level is more emphasized. Automatic algorithms, for example for mesh selection, exist, but may not be suitable for all kinds of problems. It is therefore desirable to include, in the language, means for the user to specify most aspects of the problem in order to maintain flexibility, while allowing for automatic algorithms to be used when possible and suitable.

Relating to the three levels described above, the approach taken in the design of Optimica is to extend the Modelica language with a few new language constructs corresponding to the elements of the mathematical description of the optimization problem (level I). The information included in levels II and III, however, may rather be specified by means of annotations.

3.2 Dynamic System Model

The scope of Optimica can be separated into two parts. The first part is concerned with the class of models that can be described in Modelica. Arguably, this

class is large, since very complex, non-linear and hybrid behavior can be encoded in Modelica. From a dynamic optimization perspective, the inherent complexity of Modelica models is a major challenge. Typically, different algorithms for dynamic optimization support different model structures. In fact, the key to developing efficient algorithms lies in exploiting the structure of the model being optimized. Consequently, there are different algorithms for different model structures, such as linear systems, non-linear ODEs, general DAEs, and hybrid systems. In general, an algorithm can be expected to have better performance, in terms of convergence properties and shorter execution times, if the model structure can be exploited. For example, if the model is linear, and the cost function is quadratic, the problem can be obtained very efficiently by solving a Riccati equation. On the other hand, optimization of general non-linear and hybrid DAEs is still an area of active research, see for example [3]. As a result, the structure of the model highly affects the applicability of different algorithms. The Optimica compiler presented in this paper relies on a direct collocation algorithm in order to demonstrate the proposed concept. Accordingly, the restrictions imposed on model structure by this algorithm apply when formulating the Modelica model, upon which the optimization problem is based. For example, this excludes the use of hybrid constructs, since the right hand side of the dynamics is assumed to be twice continuously differentiable. Obviously, this restriction excludes optimization of many realistic Modelica models. On the other hand, in some cases, reformulation of discontinuities to smooth approximations may be possible in order to enable efficient optimization. This is particularly important in on-line applications. The Optimica extension, as presented in this paper, could also be extended to support other algorithms, which are indeed applicable to a larger class of models.

3.3 The Dynamic Optimization Problem

The second part of the scope of Optimica is concerned with the remaining elements of the optimization problem. This includes cost functions, constraints and variable bounds. Consider the following formulation of a dynamic optimization problem:

$$\min_{u(t), p} \psi(\bar{z}, p) \quad (1)$$

subject to the dynamic system

$$F(\dot{x}(t), x(t), y(t), u(t), p, t) = 0, \quad t \in [t_0, t_f] \quad (2)$$

and the constraints

$$c_{ineq}(x(t), y(t), u(t), p) \leq 0 \quad t \in [t_0, t_f] \quad (3)$$

$$c_{eq}(x(t), y(t), u(t), p) = 0 \quad t \in [t_0, t_f] \quad (4)$$

$$c_{ineq}^p(\bar{z}, p) \leq 0 \quad (5)$$

$$c_{eq}^p(\bar{z}, p) = 0 \quad (6)$$

where $x(t) \in R^{n_x}$ are the dynamic variables, $y(t) \in R^{n_y}$ are the algebraic variables, $u(t) \in R^{n_u}$ are the control inputs, and $p \in R^{n_p}$ are parameters which are free in the optimization. In addition, the optimization is performed on the interval $t \in [t_0, t_f]$, where t_0 and t_f can be fixed or free, respectively. In addition, the initial values of the dynamic and algebraic variables may be fixed or free in the optimization. The vector \bar{z} is composed from discrete time points of the states, controls and algebraic variables; $\bar{z} = [x(t_1), \dots, x(t_{N_p}), y(t_1), \dots, y(t_{N_p}), u(t_1), \dots, u(t_{N_p})]^T$, $t_i \in [t_0, t_f]$, where N_p denotes the number of time points included in the optimization problem.

The constraints include inequality and equality path constraints, (3)-(4). In addition, inequality and equality point constraints, (5)-(6), are supported. Point constraints are typically used to express initial or terminal constraints, but can also be used to specify constraints for time points in the interior of the interval.

The cost function (1) is a generalization of a terminal cost function, $\phi(t_f)$, in that it admits inclusion of variable values at other time instants. This form includes some of the most commonly used cost function formulations. A Lagrange cost function can be obtained by introducing an additional state variable, $x_L(t)$, with the associated differential equation $\dot{x}_L(t) = L(x(t), u(t))$, and the cost function $\psi(t_f) = x_L(t_f)$. The need to include variable values at discrete points in the interior of the optimization interval in the cost function arises for example in parameter estimation problems. In such cases, a sequence of measurements, $y_d(t_i)$, obtained at the sampling instants t_i , $i \in 1 \dots N_d$ is typically available. A cost function candidate is then:

$$\sum_{i=1}^{N_d} (y(t_i) - y_d(t_i))^T W (y(t_i) - y_d(t_i)) \quad (7)$$

where $y(t_i)$ is the model response at time t_i and W is a weighting matrix.

Another important class of problems is static optimization problems on the form:

problems on the form:

$$\begin{aligned} & \min_{u,p} \phi(x, y, u, p) \\ & \text{subject to} \\ & F(0, x, y, u, p, t_s) = 0 \\ & c_{ineq}(x, u, p) \leq 0 \\ & c_{eq}(x, u, p) = 0 \end{aligned} \quad (8)$$

In this case, a static optimization problem is derived from a, potentially, dynamic Modelica model by setting all derivatives to zero. Since the problem is static, all variables are algebraic and accordingly, no transcription procedure is necessary. The variable t_s denotes the time instant at which the static optimization problem is defined.

3.4 Transcription

In this paper a direct collocation method (see for example [4]) will be used to illustrate how also the transcription step can be encoded in the Optimica extension. The information that needs to be provided by the user is then a mesh specification, the collocation points, and the coefficients of the interpolation polynomials.

4 The Optimica Extension

In this section, the Optimica extension will be presented and informally defined. The presentation will be made using the following dynamic optimization problem, based on a double integrator system, as an example:

$$\min_{u(t)} \int_0^{t_f} 1 dt \quad (9)$$

subject to the dynamic constraint

$$\begin{aligned} \dot{x}(t) &= v(t), & x(0) &= 0 \\ \dot{v}(t) &= u(t), & v(0) &= 0 \end{aligned} \quad (10)$$

and

$$\begin{aligned} x(t_f) &= 1, & v(t_f) &= 0 \\ v(t) &\leq 0.5, & -1 &\leq u(t) \leq 1 \end{aligned} \quad (11)$$

In this problem, the final time, t_f , is free, and the objective is thus to minimize the time it takes to transfer the state of the double integrator from the point $(0, 0)$ to $(1, 0)$, while respecting bounds on the velocity $v(t)$ and the input $u(t)$. A Modelica model for the double integrator system is shown in Listing 1.

In summary, the Optimica extension consists of the following elements:

```

model DoubleIntegrator
  Real x(start=0);
  Real v(start=0);
  input Real u;
equation
  der(x)=v;
  der(v)=u;
end DoubleIntegrator;

```

Listing 1: A Modelica model of a double integrator system.

- A new specialized class: `optimization`
- New attributes for the built-in type `Real`: `free` and `initialGuess`.
- A new function for accessing the value of a variable at a specified time instant
- Class attributes for the specialized class `optimization`: `objective`, `startTime`, `finalTime` and `static`
- A new section: `constraint`
- Inequality constraints
- An annotation for providing transcription information

4.1 A New Specialized Class

It is convenient to introduce a new specialized class, called `optimization`, in which the proposed Optimica-specific constructs are valid. This approach is consistent with the Modelica language, since there are already several other specialized classes, e.g., `record`, `function` and `model`. By introducing a new specialized class, it also becomes straightforward to check the validity of a program, since the Optimica-specific constructs are only valid inside an `optimization` class. The `optimization` class corresponds to an optimization problem, static or dynamic, as specified in Section 3.3. Apart from the Optimica-specific constructs, an `optimization` class can also contain component and variable declarations, local classes, and equations.

It is not possible to declare components from `optimization` classes in the current version of Optimica. Rather, the underlying assumption is that an `optimization` class defines an optimization problem, that is solved off-line. An interesting extension would, however, be to allow for `optimization` classes to be

instantiated. With this extension, it would be possible to solve optimization problems, on-line, during simulation. A particularly interesting application of this feature is model predictive control, which is a control strategy that involves on-line solution of optimization problems during execution.

As a starting-point for the formulation of the optimization problem (9)-(11), consider the optimization class:

```

optimization DIMinTime
  DoubleIntegrator di;
end DIMinTime;

```

This class contains only one component representing the dynamic system model, but will be extended in the following to incorporate also the other elements of the optimization problem.

4.2 Attributes for the Built-in Type Real

In order to superimpose information on variable declarations, two new attributes are introduced for the built-in type `Real`². Firstly, it should be possible to specify that a variable, or parameter, is free in the optimization. Modelica parameters are normally considered to be fixed after the initialization step, but in the case of optimization, some parameters may rather be considered to be free. In optimal control formulations, the control inputs should be marked as free, to indicate that they are indeed optimization variables. For these reasons, a new attribute for the built-in type `Real`, `free`, of boolean type is introduced. By default, this attribute is set to `false`.

Secondly, an attribute, `initialGuess`, is introduced to enable the user to provide an initial guess for variables and parameters. In the case of free optimization parameters, the `initialGuess` attribute provides an initial guess to the optimization algorithm for the corresponding parameter. In the case of variables, the `initialGuess` attribute is used to provide the numerical solver with an initial guess for the entire optimization interval. This is particularly important if a simultaneous or multiple-shooting algorithm is used, since these algorithms introduce optimization variables corresponding to the values of variables at discrete points over the interval. Notice that such initial guesses may be needed both for control and state variables. For such variables, however, the proposed strategy for providing initial guesses may sometimes be inadequate.

²The same attributes may be introduced for the built-in type `Integer`, in order to support also variables of type `Integer` in the optimization formulation

In some cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler. In the double integrator example, the control variable u is a free optimization variable, and accordingly, the `free` attribute is set to `true`. Also, the `initialGuess` attribute is set to 0.0.

```

optimization DIMinTime
  DoubleIntegrator di(u(free=true,
                       initialGuess=0.0));
end DIMinTime;

```

4.3 A Function for Accessing Instant Values of a Variable

An important component of some dynamic optimization problems, in particular parameter estimation problems where measurement data is available, is variable access at discrete time instants. For example, if a measurement data value, y_i , has been obtained at time t_i , it may be desirable to penalize the deviation between y_i and a corresponding variable in the model, evaluated at the time instant t_i . In Modelica, it is not possible to access the value of a variable at a particular time instant in a natural way, and a new construct therefore has to be introduced.

All variables in Modelica are functions of time. The variability of variables may be different—some are continuously changing, whereas others can change value only at discrete time instants, and yet others are constant. Nevertheless, the value of a Modelica variable is defined for all time instants within the simulation, or optimization, interval. The time argument of variables are not written explicitly in Modelica, however. One option for enabling access to variable values at specified time instants is therefore to associate an implicitly defined function with a variable declaration. This function can then be invoked by the standard Modelica syntax for function calls, $y(t_i)$. The name of the function is identical to the name of the variable, and it has one argument; the time instant at which the variable is evaluated. This syntax is also very natural since it corresponds precisely to the mathematical notation of a function. Notice that the proposed syntax $y(t_i)$ makes the interpretation of such an expression context dependent. In order for this construct to be valid in standard Modelica, y must refer to a function declaration. With the proposed extension, y may refer either to a function declaration or a variable declaration. A compiler therefore needs to classify an expression $y(t_i)$ based on the context, i.e.,

what function and variable declarations are visible. An alternative syntax would have been to introduce a new built-in function, that returns the value of a variable at a specified time instant. While this alternative would have been straightforward to implement, the proposed syntax has the advantages of being easier to read and that it more closely resembles the corresponding mathematical notation. This feature of Optimica is used in the constraint section of the double integrator example, and is described below.

4.4 Class Attributes

In the optimization formulations (1)-(6) and (8), there are elements that occur only once, i.e., the cost function and the optimization interval in (1)-(6), and in the static case (8), only the cost function. These elements are intrinsic properties of the respective optimization formulations, and should be specified, once, by the user. In this respect the cost function and optimization interval differ from, for example, constraints, since the user may specify zero, one or more of the latter.

One option for providing this kind of information is to introduce a built-in class, call it `Optimization`, and require that all optimization classes inherit from `Optimization`. Information about the cost function and optimization interval may then be given as modifications of components in this built-in class:

```

optimization DIMinTime
  extends Optimization(
    objective=cost(finalTime),
    startTime=0,
    finalTime(free=true,initialGuess=1));
  Real cost;
  DoubleIntegrator di(u(free=true,
                       initialGuess=0.0));
equation
  der(cost) = 1;
end DIMinTime;

```

Here, `objective`, `startTime` and `finalTime` are assumed to be components located in `Optimization`, whereas `cost` is a variable which is looked up in the scope of the optimization class itself. Notice also how the cost function, `cost`, has been introduced, and that the `finalTime` attribute is specified to be free in the optimization. This approach of inheriting from a built-in class has been used previously, in the tool Mosilab [11], where the Modelica language is extended to support statecharts. In the statechart extension, a new specialized class, `state`, is introduced, and properties of a state class (for example whether the state is an initial state) can be specified by inherit-

ing from the built-in class `State` and applying suitable modifications.

The main drawback of the above approach is its lack of clarity. In particular, it is not immediately clear that `Optimization` is a built-in class, and that its contained elements represent intrinsic properties of the optimization class, rather than regular elements, as in the case of inheritance from user or library classes. To remedy this deficiency, the notion of *class attributes* is proposed. This idea is not new, but has been discussed previously within the Modelica community. A class attribute is an intrinsic element of a specialized class, and may be modified in a class declaration without the need to explicitly extend from a built-in class. In the Optimica extension, four class attributes are introduced for the specialized class `optimization`. These are `objective`, which defines the cost function, `startTime`, which defines the start of the optimization interval, `finalTime`, which defines the end of the optimization interval, and `static`, which indicates whether the class defines a static or dynamic optimization problem. The proposed syntax for class attributes is shown in the following optimization class:

```
optimization DIMinTime (
    objective=cost(finalTime),
    startTime=0,
    finalTime(free=true,initialGuess=1))
Real cost;
DoubleIntegrator di(u(free=true,
    initialGuess=0.0));
equation
    der(cost) = 1;
end DIMinTime;
```

The default value of the class attribute `static` is `false`, and accordingly, it does not have to be set in this case. In essence, the keyword `extends` and the reference to the built-in class have been eliminated, and the modification construct is instead given directly after the name of the class itself. The class attributes may be accessed and modified in the same way as if they were inherited.

4.5 Constraints

Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. In Modelica, initial equations, equations, and algorithms are specified in separate sections, within a class

body. A reasonable alternative for specifying constraints is therefore to introduce a new kind of section, `constraint`. Constraint sections are only allowed inside an optimization class, and may contain equality, inequality as well as point constraints. In the double integrator example, there are several constraints. Apart from the constraints specifying bounds on the control input u and the velocity v , there are also terminal constraints. The latter are conveniently expressed using the mechanism for accessing the value of a variable at a particular time instant; `di.x(finalTime)=1` and `di.v(finalTime)=0`. In addition, bounds may have to be specified for the `finalTime` class attribute. The resulting optimization formulation may now be written:

```
optimization DIMinTime (
    objective=cost(finalTime),
    startTime=0,
    finalTime(free=true,initialGuess=1))
Real cost;
DoubleIntegrator di(u(free=true,
    initialGuess=0.0));
equation
    der(cost) = 1;
constraint
    finalTime>=0.5;
    finalTime<=10;
    di.x(finalTime)=1;
    di.v(finalTime)=0;
    di.v<=0.5;
    di.u>=-1; di.u<=1;
end DIMinTime;
```

4.6 Annotations for Specification of the Transcription Scheme

The transcription scheme used to transform the infinite-dimensional dynamic optimization problem into a finite-dimensional approximate problem usually influences the properties of the numerical solution. Nevertheless, transcription information can be considered to be complimentary information, that is not part of the mathematical definition of the optimization problem itself. Also, transcription information is closely related to particular numerical algorithms. It is therefore reasonable not to introduce new language constructs, but rather new annotations for specification of transcription schemes. This solution is also more flexible, which is important in order easily accommodate transcription schemes corresponding to algorithms other than the direct collocation method currently supported.

Following the guidelines for vendor-specific annota-

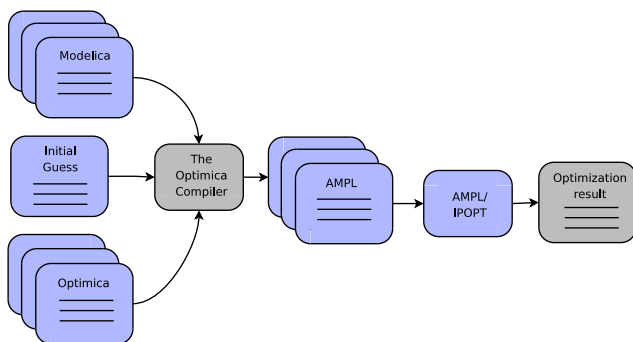


Figure 1: The transformation from Modelica/Optimica code to optimization result.

tions in the specification of Modelica 3.0 [13, p. 147], a hierarchical annotation for supplying the information needed to specify a direct collocation method based on interpolation polynomials has been introduced. This annotation is defined by the following Modelica record:

```

record DirectCollocationInterpolationPolynomials
  parameter Real mesh[:];
  parameter Real collocationPoints[:];
  parameter Real
    polynomialCoefficientsAlgebraic[:];
  parameter Real
    polynomialCoefficientsDynamic[:];
end DirectCollocationInterpolationPolynomials;

```

This annotation enables the user to influence the particular properties of the corresponding transcription scheme. For additional details, see [1].

5 The Optimica Compiler

A new Modelica compiler, entitled the JModelica compiler is currently under development [2, 1]. The compiler is developed in the compiler construction framework JastAdd, see [9], and in Java. One of the primary targets of the JModelica compiler is to provide an extensible compiler, which is suitable for modular implementation of new language features. A fundamental design concept is that of *modular extensibility*, which enables the core JModelica compiler to be kept intact, since new extensions may be implemented fully modularized.

A prototype implementation of the JModelica compiler, that also supports the Optimica extension has been developed. The extended compiler will be referred to as the Optimica compiler in the following. In terms of the front-end, the compiler supports a subset of Modelica, and an early version of Optimica. The

syntax of Optimica that is supported by this compiler is different than the one presented in this paper, although the functionality is essentially the same. The new, improved syntax and semantics that have been presented in this paper, were defined based on the comments and experiences from the users of the very first version of Optimica. A new version of the Optimica compiler, supporting the revised Optimica syntax is currently under development, with the intention of replacing the initial prototype.

5.1 Code Generation to AMPL

One of the main features of the Optimica compiler is that it performs automatic transcription of continuous variables, using a direct collocation method. The user is thus relieved from the burden of encoding the collocation equations, which is a tedious and error-prone procedure. Whereas the prototype version of the Optimica compiler supported one particular collocation scheme, future versions will support the annotation introduced above to specify the transcription method.

In order to solve the transcribed optimization problem by means of a numerical algorithm, the Optimica compiler generates AMPL [7] code. The transcribed problem is purely static, and can therefore be encoded using the constructs available in AMPL. The AMPL representation of the optimization problem can be viewed as an additional intermediate representation format. The purpose of using AMPL is twofold. Firstly, AMPL provides an additional debugging level, that is very useful during compiler development. In particular, the AMPL tool offers a shell, where variables and constraints can be inspected. Secondly, the AMPL solver interface provides solvers with sparsity information, as well as first and second order derivatives. This information may be essential for performance and convergence of a numerical optimization algorithm. The numerical algorithm IPOPT [14] has been used to solve the non-linear program resulting from the transcription procedure. The result is then written to file for further analysis or implementation. See Figure 1 for an illustration of the transformation steps involved when using the Optimica compiler and AMPL to solve a dynamic optimization problem.

6 Summary and Conclusions

In this paper an extension of the Modelica language, Optimica, that enables high-level formulation of dynamic optimization problems, has been presented. The

Optimica extension enables the user to specify important elements of a dynamic optimization problem such as cost functions, constraints and optimization interval. The dynamic model, upon which the dynamic optimization problem is based, is expressed using standard Modelica. Optimica also supports an annotation that enables the user to specify the properties of a transcription method, based on direct collocation. Because of these properties, Optimica supports formulation of dynamic optimization problems, using high-level constructs, both at the mathematical level and at the numerical transcription level.

A prototype implementation of the Optimica compiler has been used in the work on start-up optimization of a plate reactor [8], in two master's thesis projects (see [5] and [10]) and in the PhD course "Optimization-Based Methods and Tools in Control", that was given at the Department of Automatic Control, Lund University in September 2007.

An important objective of the JModelica compiler is to offer a modularly extensible Modelica compiler. In this respect, the experiences and results from developing the Optimica extension are very promising. In particular, the coding effort needed to implement the extension of the compiler front-end, including extension of the name analysis framework and the flattening algorithm, was very moderate.

References

- [1] J. Åkesson. Tools and Languages for Optimization of Large-Scale Systems. PhD thesis ISRN LUTFD2/TFRT--1081--SE, Department of Automatic Control, Lund University, Sweden, November 2007.
- [2] J. Åkesson, T. Ekman, and G. Hedin. "Development of a Modelica compiler using JastAdd." In *Seventh Workshop on Language Descriptions, Tools and Applications*, Braga, Portugal, March 2007.
- [3] P. Barton and C. K. Lee. "Modeling, simulation, sensitivity analysis, and optimization of hybrid systems." *ACM Transactions on Modeling and Computer Simulation*, **12:4**, 2002.
- [4] L. Biegler, A. Cervantes, and A. Wächter. "Advances in simultaneous strategies for dynamic optimization." *Chemical Engineering Science*, **57**, pp. 575–593, 2002.
- [5] H. Danielsson. "Vehicle path optimisation." Master's Thesis ISRN LUTFD2/TFRT--5797--SE, Department of Automatic Control, Lund University, Sweden, June 2007.
- [6] Dynasim AB. "Dynasim AB Home Page." 2007. <http://www.dynasim.se>.
- [7] R. Fourer, D. Gay, and B. Kernighan. *AMPL – A Modeling Language for Mathematical Programming*. Brooks/Cole — Thomson Learning, 2003.
- [8] S. Haugwitz, J. Åkesson, and P. Hagander. "Dynamic optimization of a plate reactor start-up supported by Modelica-based code generation software." In *Proceedings of 8th International Symposium on Dynamics and Control of Process Systems*, Cancun, Mexico, June 2007.
- [9] G. Hedin and E. Magnusson. "JastAdd: an aspect-oriented compiler construction system." *Science of Computer Programming*, **47:1**, pp. 37–58, 2003.
- [10] H. Hultgren and H. Jonasson. "Automatic calibration of vehicle models." Master's Thesis ISRN LUTFD2/TFRT--5794--SE, Department of Automatic Control, Lund University, Sweden, June 2007.
- [11] C. Nytsch-Geusen. "MosiLab Home Page." 2007. <http://www.mosilab.de/>.
- [12] Process Systems Enterprise. "gPROMS Home Page." 2007. <http://www.psenderprise.com/gproms/index.html>.
- [13] The Modelica Association. "Modelica – a unified object-oriented language for physical systems modeling, language specification, version 3.0." Technical Report, Modelica Association, 2007.
- [14] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming." *Mathematical Programming*, **106:1**, pp. 25–58, 2006.

Session 1c

Automotive Applications



Detailed Simulation of Turbocharged Engines with Modelica

John J. Batteh Charles E. Newman

Ford Motor Company, Research and Advanced Engineering, USA

jbatteh@ford.com, cnewman@ford.com

Abstract

This paper describes the development and application of detailed models for the simulation of turbocharged spark-ignited engines in Modelica. Following a brief overview of previously-published modeling capabilities, a new engine architecture that provides the flexibility required for simulating boosted systems is detailed. Techniques for turbocharger modeling are discussed followed by sample steady state and transient simulations that illustrate potential model usage in design and control applications.

Keywords: cycle simulation; turbocharging; engine; thermodynamics

1 Introduction

The convergence of increasingly-stringent fuel economy and CO₂ emissions standards and an overall increase in the awareness and impact of global warming trends have led to increased focus on advanced vehicle concepts for improved fuel economy. Given the historical growth in market share of large trucks and sport utility vehicles in the US shown in Figure 1 [1], the focus on improved fuel economy is especially acute. Vehicle fuel economy is clearly a system attribute that is affected by a myriad of different factors, including powertrain system configuration, vehicle weight, aerodynamic drag, rolling resistance, controls and calibration features, and various component efficiencies in the system. While OEMs are exploring opportunities in all aspects of the fuel economy picture, one area of continued focus is on the fuel consumption of the primary powerplant.

A potential opportunity for increasing fuel economy of spark-ignited engines is by turbocharging in combination with engine downsizing. The first patent [2] for a turbocharger on an internal combustion engine was filed in 1905 by Alfred Buchi, a Swiss engineer. Figure 2 shows a sample schematic of a turbocharged engine [3]. The exhaust

from the engine is routed through a turbine where exhaust energy is extracted to drive the compressor. The compressed air is typically fed through an inter-cooler before being routed to the engine. When compared with naturally-aspirated engines, turbocharged engines have increased volumetric efficiency and specific power output thereby enabling engine downsizing. Benefits from engine downsizing include reduced pumping (throttling) losses for part load operation, potential friction reductions, and also potential reductions in powertrain system weight. It should be noted that turbocharging does not come without cost. A few commonly-cited disadvantages of turbocharged engines are increased backpressure to the engine, hardware and controls complexity and cost, and the potential for "turbo lag", broadly defined as the time required from the initial driver throttle demand to spin up the turbo, increase the boost, and deliver the requested torque.

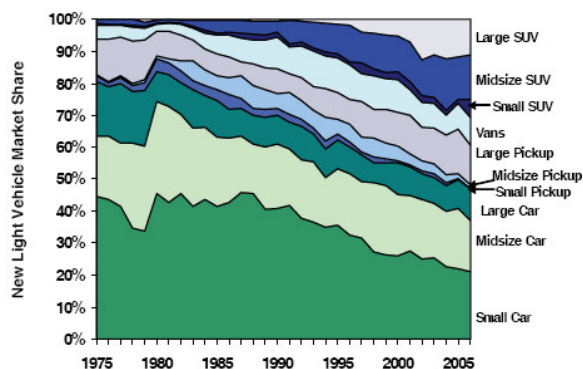


Figure 1. US light vehicle market share, 1975-2006 [1]

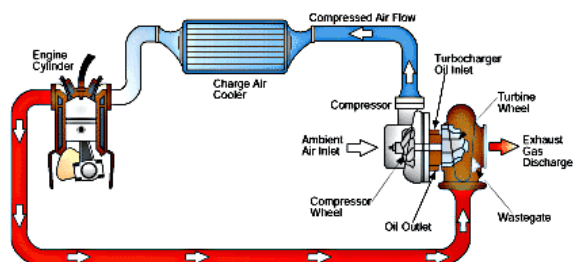


Figure 2. Turbocharged engine schematic [3]

The turbocharger introduces strong feedback between the exhaust and intake systems. Coupled with the different time scales in the engine system, robust design and control of turbocharged engine systems is challenging, even more so with an intense scrutiny on fuel economy benefits. Thus analytic capability for detailed simulation of turbocharged engines is a key enabler for upfront powertrain system design. Potential simulation applications include analytic turbocharger matching and optimization, advanced engine concept assessment, and assessment of transient turbocharger performance.

This paper describes the development and application of detailed models for simulation of turbocharged spark-ignited engines in Modelica [4]. Following a brief overview of previously-published engine cycle simulation capability, new architecture changes are detailed that allow for configurable, efficient modeling of turbocharged engines. Modeling of the turbochargers is also discussed, and some sample steady state and transient results are shown.

2 Engine Model Architecture

Detailed cycle simulation modeling and applications have been discussed in depth in previous publications [5]-[7]. These publications describe an engine model architecture for flexible modeling of the intake, mixture preparation, combustion, and exhaust processes for spark-ignited engines. The crankangle-resolved model includes submodels for breathing past the intake and exhaust valves based on discharge coefficients as a function of valve lift, flow-based turbulence generation and dissipation, mixture preparation and injection dynamics, predictive combustion with laminar and turbulent flame propagation, and heat transfer and thermal warm-up. These models have been used in both steady-state and transient applications for design optimization and robustness, performance, fuel economy, and cold start.

2.1 Restructuring

Previous applications of the engine model were focused on naturally aspirated applications. The existing engine architecture divided the engine into cylinders with each individual cylinder model containing the intake, exhaust, and combustion chamber submodels. The architecture supported both single and multi cylinder applications via engine templates with `replaceable` cylinder models. Inside a given engine template, the instantiated cylinders were wired to the external connectors for the crank-

shaft, engine block, and intake and exhaust ambient reservoirs.

The existing architecture provided highly flexible for naturally aspirated engines but did not provide the necessary configurability for boosted applications. Figure 3 shows the new, restructured engine model architecture. The new structure divides the engine along the head (`intake_exhaust_system` component) and block (`bottom` component). The connection between the head and block is an array based on the number of modeled cylinders. The head contains the model of the intake and exhaust system such as the throttle, plenum, and individual cylinder heads, which contain the fuel injectors and intake/exhaust ports and valves. The block component consists of the individual combustion chambers which primarily contain the respective cylinder volumes and combustion models. One addition to the engine structure is a `replaceable` boost device model situated between the intake and exhaust reservoir connectors and the head component. The constraining class for this component is of sufficient generality that it can be replaced by a class which can simulate naturally aspirated, supercharged, turbocharged, or turbocompounded behavior.

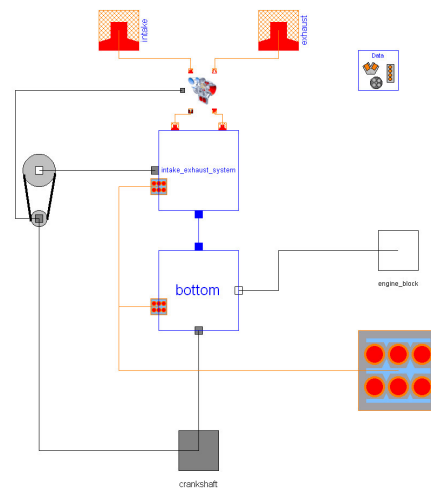


Figure 3. New engine model structure

2.2 Surrogate Modeling

Single cylinder models are often used to represent multi-cylinder engines due to their computational efficiency. While this representation is more appropriate for steady-state applications and some transient applications with prescribed intake and exhaust conditions, it is typically not appropriate for turbocharged applications where the transient blow-

down pulses from the cylinders provide the exhaust energy that drives the turbocharger. Crankangle-resolved representation of turbocharged engines requires the modeling of the filling and emptying dynamics of the intake and exhaust manifolds to accurately represent the downstream compressor and upstream turbine conditions respectively.

In an effort to retain the computational efficiency of single cylinder modeling for turbocharged engines, a new structure is introduced consisting of both primary and surrogate cylinder-head representations. The detailed breathing calculations are performed in the primary cylinder head, which is connected to the detailed combustion model. The resulting flows of chemical species and energy from the breathing calculations in the primary cylinder are then replicated in the surrogate cylinder head representations at the appropriate phasing as surrogates for the contributions of the missing cylinders. The implicit assumption is that the manifold conditions are quasi-steady on the time scale of a single, complete firing cycle of the engine.

Figure 4 shows the surrogate flow structure. Figure 4a shows an engine head model with a single cylinder intake system mimicking a multicylinder engine. The intake and exhaust surrogate models are positioned between the manifolds and the head model for the primary cylinder. Figure 4b shows a single instance of the surrogate model. The primary flow path is broken by a flow sensor that is used by the surrogate flow source that is instantiated in parallel.

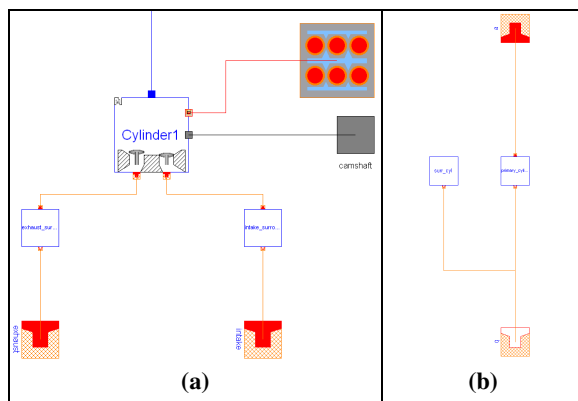


Figure 4. Surrogate flow structure

Figure 5 illustrates the surrogate flow concept. Figure 5a depicts a surrogate flow representation for the intake of an I6 engine. There is a single primary flow with five replicated and phased surrogate flows. For clarity, only the primary flow is shown from the first cycle followed by all the flows in the second

cycle. The total flow is the superposition of all the flows. Figure 5b shows a surrogate exhaust flow representation for a single bank of a V6 engine. The total exhaust flow is not pictured as it obscured the ability to see clearly the individual surrogate flows. The dynamic exhaust events from the individual cylinders that are used to drive the turbine are clearly captured. Note that the phasing changes appropriately based on the number of replicated cylinders.

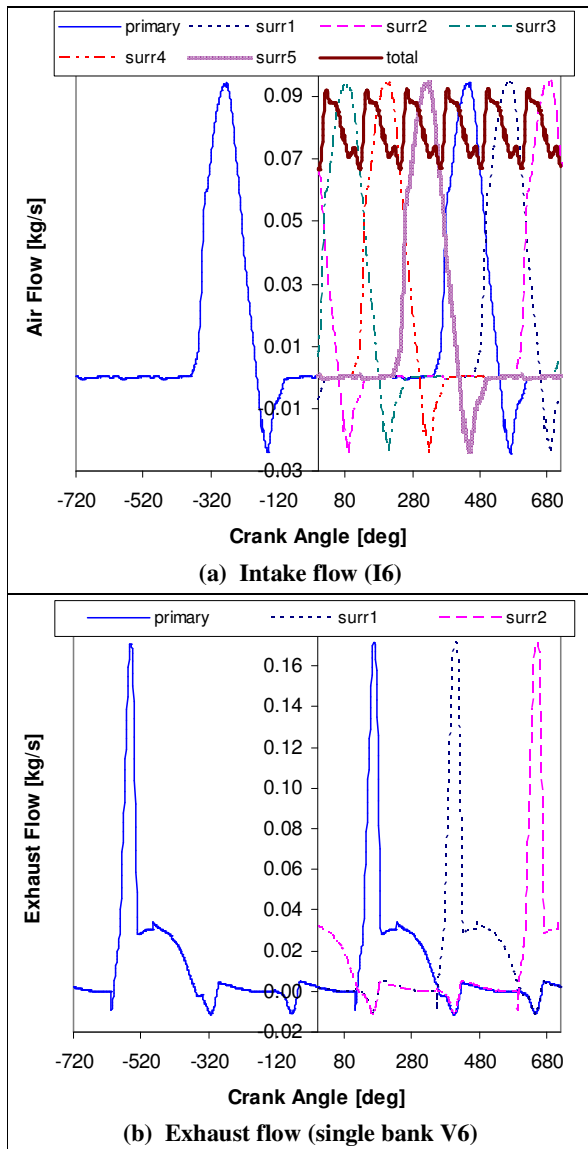


Figure 5. Sample intake (a) and exhaust (b) flows

Several alternatives for the surrogate flow calculations were implemented in Modelica. The alternatives differ in the way in which the surrogate flows are calculated and provide slightly different numerical results. The traces shown in Figure 5 are from

the DelayedSurrogateFlow model. This model uses the built-in `delay` operator to phase the mass flow rates for the surrogate cylinders based on the primary cylinder calculation. It is worth noting that this implementation yields numerical Jacobians in Dymola (and in the authors' opinion should not).

3 Turbocharger Modeling

In addition to the engine restructuring to support inclusion of boost device models, various boost device models were implemented in Modelica. Figure 6 shows a model for an exhaust-driven turbocharger. The ConfigurableTurboCompressor model provides a template for turbocharger modeling. It consists of a turbine component connected to the compressor component by the turbine_shaft. There is also a wastegate component on the turbine side and an intercooler component on the compressor side. Extensive use of replaceable models allow for flexibility in configuring the template to simulate specific hardware.

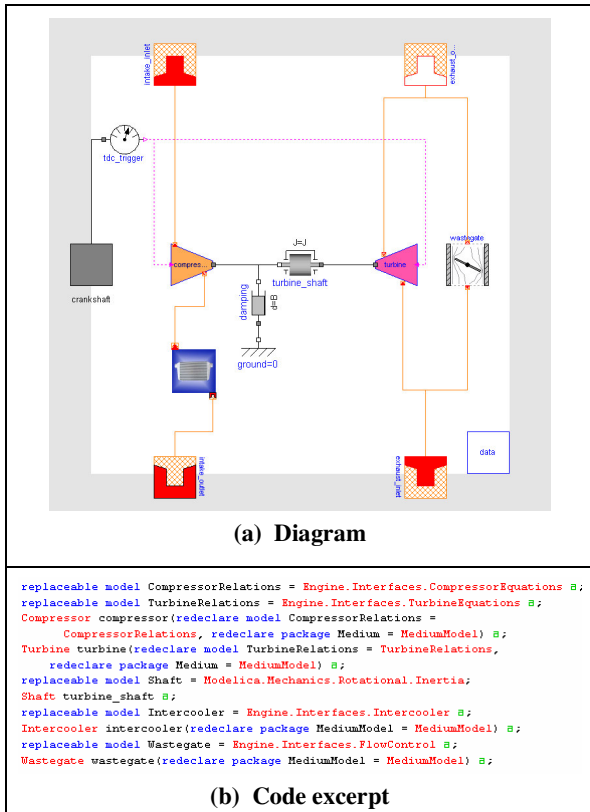


Figure 6. Turbo model

Detailed modeling of turbocharger behavior based on geometric information typically requires CFD-type simulations. For lumped systems models, steady-state mapped data, typically provided by the component supplier from gas stand testing, is often used to simulate component model behavior. The mapped data for the turbine and compressor consists of mass flow rate and efficiency data over a range of shaft speeds and pressure ratios [8]. Figure 7 shows a sample compressor efficiency map with annotations showing the various features of the map (*i.e.* surge line, choke line, efficiency islands, speed lines, *etc.*) [9].

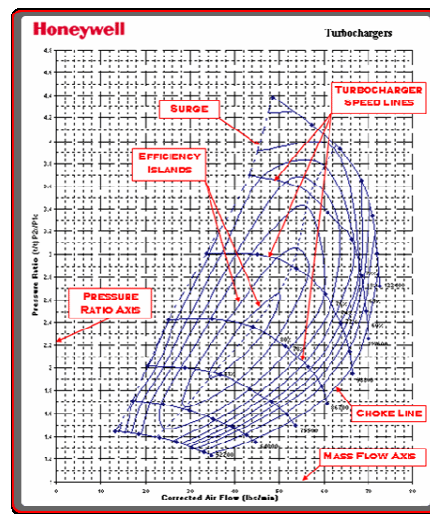


Figure 7. Sample compressor efficiency map [9]

Typically the mapped data exists for a rather limited range of speeds and pressure ratios and must be extended analytically to ensure model robustness. There are a variety of ways to implement and/or fit the map data for component modeling [10]. The following discussion and figures provide some sample results from the fitting procedure used by the authors.

To account for differences in inlet conditions, turbine map data is often provided in reduced form. The map data gives reduced mass flow and efficiency as a function of reduced speed and pressure ratio as defined in the following equations:

$$N_r = \frac{N}{\sqrt{T_{inlet}}} \tag{1}$$

$$\dot{m}_r = \frac{\dot{m}\sqrt{T_{inlet}}}{P_{inlet}} \tag{2}$$

where N is the shaft speed in RPM, \dot{m} is the flow rate through the turbine in kg/s, and the inlet pressure

and temperature conditions are denoted by P_{inlet} and T_{inlet} , respectively. Blade speed ratio (BSR) is defined as the blade speed divided by the isentropic enthalpy drop across the turbine and can be computed as follows [11]:

$$BSR = \frac{\frac{2\pi N}{60} \left(\frac{D}{2}\right)}{\left[2h_{in} \left(1 - PR^{\frac{1-\gamma}{\gamma}}\right)\right]^{1/2}} \quad (3)$$

where D is turbine diameter, h_{in} is the inlet enthalpy, and PR is the pressure ratio across the turbine. Note that BSR is an independent variable that combines both the pressure ratio and the shaft speed. In an attempt to collapse the data onto a single line to facilitate fitting, efficiencies and reduced mass flow rates are normalized. The normalized variables can then be fit based on a normalized blade speed ratio as shown in Figure 8.

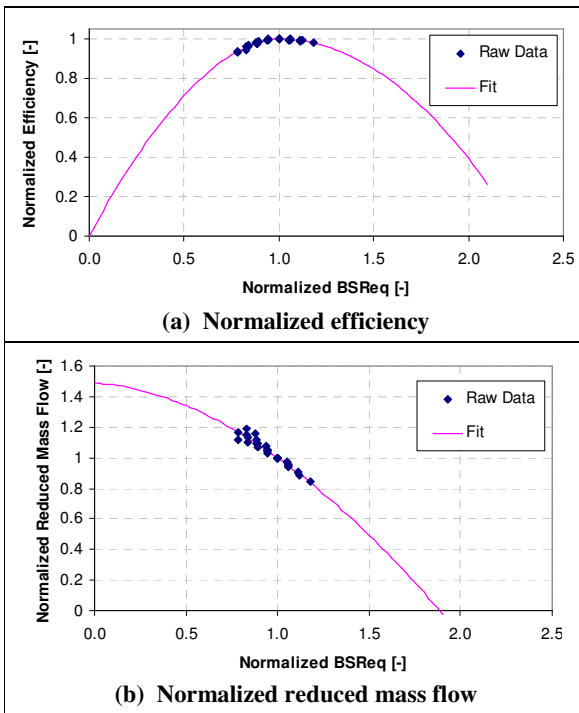


Figure 8. Sample turbine fits for normalized variables

Compressor map data is often corrected to reference conditions to account for differences in inlet conditions.

$$N_c = \frac{N}{\sqrt{\frac{T_{inlet}}{T_{reference}}}} \quad (4)$$

$$\dot{m}_c = \frac{\dot{m} \sqrt{\frac{T_{inlet}}{T_{reference}}}}{\frac{P_{inlet}}{P_{reference}}} \quad (5)$$

where $P_{reference}$ and $T_{reference}$ denote the reference conditions. Similar techniques to those described previously for the turbine can be used to fit the normalized compressor efficiency. A tabular implementation is used for the corrected mass flow data as a function of shaft speed and pressure ratio.

To facilitate the modeling of new turbocharger hardware, an external tool has been developed to generate the fits to the mapped data. Given the raw map data from the supplier, the tool calculates the various required fit coefficients for efficiency and flow rate using least square regression. As shown in the code in Figure 6b, the `replaceable` models for `CompressorRelations` and `TurbineRelations` are used to specify the mapped component behavior. Base classes for various types of raw data (*i.e.* corrected, reduced, mass flow, volume flow) and fitting techniques have been created. By extending from the appropriate base class and providing the fit coefficients, the component map for a given piece of hardware can be defined and selected for use in the `ConfigurableTurboCompressor`.

4 Simulation Results

The new engine architecture and turbocharger modeling capability in conjunction with predictive combustion cycle simulation provide a tool for up-front assessment of advanced engine concepts. Simulation results from a few sample applications are provided. The simulations were performed using Dymola [12].

4.1 Steady State

Early concept assessment typically occurs on an engine dynamometer long before vehicle work begins. These assessments are usually steady state for performance, fuel economy, and calibration. Surrogate hardware is often used prior to vehicle hardware availability thereby necessitating a configurable modeling environment for maximum flexibility.

Figure 9 compares the results from a simulated load sweep at a fixed engine speed for two different turbocharged engine concepts. The model was initially calibrated based on experimental data from an

early hardware iteration of Engine A. Following a major hardware update, the model was updated to the latest hardware level, and the original calibration was validated via prediction at a different engine operating condition. The percent difference between the model prediction and the experimental data is shown in Table 1 for various pressure, temperature, flow, and combustion statistics. The model agrees well with the experimental data. The model predictions in Figure 9 are purely analytic based on virtual hardware changes for Engine A and concept Engine B over operating conditions which were only simulated. The predicted fuel consumption for Engine B is roughly 3-4% less than that of Engine A. Figure 9b shows the steady state shaft speeds for the two engine concepts.

Table 1. Model validation (depicted as percent difference between model prediction and experimental data)

		%err
airflow	kg/s/cyl	0.078212
BMEP	bar	-0.051359
gIMEP	bar	-0.241572
PMEP	bar	
ISFC	g/kW.h	0.430916
BSFC	g/kW.h	0.240178
burn010	deg	-0.662252
ca50	degATDC	1.421053
caPmax	degATDC	2.137423
MAP	kPa	-0.676617
CompoutP	kPa	-0.620951
CompoutT	K	-0.393695
ICoutletT	K	-0.099168
TurbinP	kPa	3.410164
TurbinT	K	0.095168

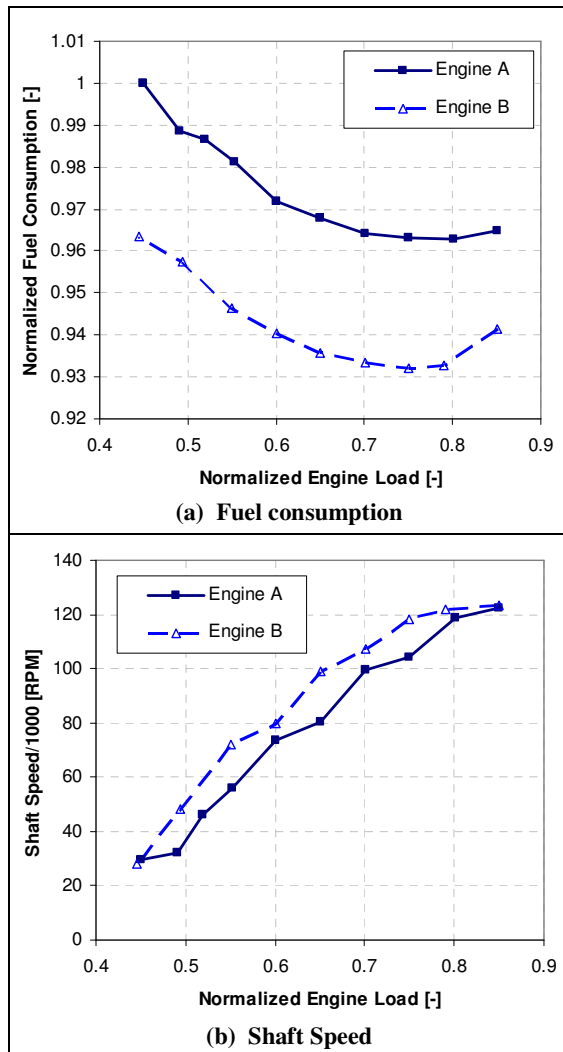


Figure 9. Load sweep at a fixed speed

4.2 Transient

In addition to steady state characterization, transient performance metrics play a crucial role in concept assessment. The ability to provide analytic assessments of transient response is a key enabler for upfront powertrain system design and optimization. In particular, transient response metrics are especially important in turbocharged engine applications to ensure robust hardware and control system design to mitigate the impact of any potential turbo lag issues.

Figure 10 shows the results for a simulated throttle transient with Engine A. The simulations were run at a fixed engine speed. As the throttle opens, the engine load quickly increases as the manifold pressure approaches compressor outlet pressure. The resulting rise in exhaust mass flow drives the turbine shaft to higher speeds, producing additional boost and increasing the load even further. The transient behavior results from the inertia of the turbocharger shaft, filling and emptying of the intake and exhaust manifolds, turbocharger performance dynamics, intercooler dynamics, and combustion phasing dynamics. Given the highly-coupled nature of turbocharged systems, a transient, physical model is an extremely valuable tool for understanding the various feedback mechanisms and key parameters for robust design and system control.

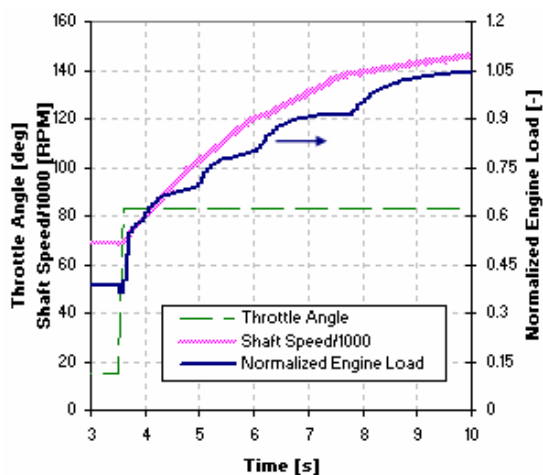


Figure 10. Throttle transient, Engine A

5 Conclusions

Development and implementation of a new engine architecture in Modelica for the detailed simulation of turbocharged, spark-ignited engines has been presented. In conjunction with previously-developed capability for predictive engine cycle simulation, the models provide a highly-capable platform for analytic, upfront design assessment and optimization for turbocharged engines. The model predictions have been validated with experimental data, and the results from several sample applications provide some insight into potential model usage.

References

- [1] Davis, S.C. and Diegel, S.W., 2007, "Transportation Energy Data Book: Edition 26", ORNL-6978, http://cta.ornl.gov/data/tedb26/Edition26_Full_Doc.pdf
- [2] Buchi, Alfred, 1905, Patent No. 204630 from the Imperial Patent Office of the German Reich.
- [3] Nice, Karim, 2007, "How Turbochargers Work", <http://auto.howstuffworks.com/turbo1.htm>
- [4] Modelica Association, 2006, "Modelica Language Specifications (Version 2.2.1)", <http://www.modelica.org>
- [5] Newman, C., Batteh, J., and Tiller, M., 2002, "Spark-Ignited-Engine Cycle Simulation in Modelica", *2nd International Modelica Conference Proceedings*, pp. 133-142, http://modelica.org/Conference2002/papers/p17_Newman.pdf
- [6] Batteh, J., Tiller, M., and Newman, C., 2003, "Simulation of Engine Systems in Modelica", *3rd International Modelica Conference Proceedings*, pp. 139-148, http://www.modelica.org/events/Conference2003/papers/h34_Batteh.pdf
- [7] Batteh, J.J., Tiller, M. and Goodman, A., 2005, "Monte Carlo Simulations for Evaluating Engine NVH Robustness", *Proceedings of the 4th International Modelica Conference*, p. 385-392, http://www.modelica.org/events/Conference2005/online_proceedings/Session5/Session5a1.pdf
- [8] Society of Automotive Engineers, 1995, *Turbocharger Gas Stand Test Code*, SAE Standard J1826, SAE Press: Warrendale, PA.
- [9] Garrett, Turbo Tech 103, 2007, http://www.turbobygarrett.com/turbobygarrett/tech_center/turbo_tech103.html.
- [10] Moraal, P. and Kolmanovsky, I., 1999, "Turbocharger Modeling for Automotive Control Applications", SAE 1999-01-0908, Society of Automotive Engineers.
- [11] Heywood, J.B., 1988, *Internal Combustion Engine Fundamentals*. McGraw-Hill.
- [12] Dymola. Dynasim AB, Lund, Sweden, <http://www.dynasim.com>.

Thermal Modelling of an Automotive Nickel Metall Hydrid Battery in Modelica using Dymola

Helmut Oberguggenberger Dragan Simic
Arsenal Research
Giefinggasse 2, 1210 Vienna, Austria

Abstract

This paper deals with the thermal modelling of an automotive nickel metall hydrid battery. The thermal modelling will be done in two different approaches. The result of the distinct approaches will be the same, though.

The thermal models are implemented in *Modelica* simulation language and simulated using the *Dymola* simulation environment, [1].

Thermal and electrical measurements have been carried out to validate the simulation results of the thermal modelling and will be presented in this paper.

Keywords: simulation, modelling, nickel metall hydrid battery, validation

1 Simple thermal cell models

The first approach is the thermal modelling of a nickel metall hydrid battery package taking into account the geometry and temperature distribution during the operation of a single cell.

In this paper a thermal model of a cubical-shaped package of a battery will be modelled, simulated and evaluated. The thermal battery model comprises algebraic and ordinary differential equations. All components of the thermal battery model are taken from the *ModelicaStandardLibrary*, such as *Modelica.Thermal.HeatTransfer*.

The thermal model of the battery is modelled by means of discrete volume elements. In this model the coefficients of heat transfer for each discrete volume are calculated. The cell model, `cell` in figure 1, represents the thermal model with all inner thermal behaviors. The heat flow inside the cell in all three directions was implemented. The model was parametrized using geometrical and thermal measured data of the cell, such as length, width, thermal conductivity, density of the

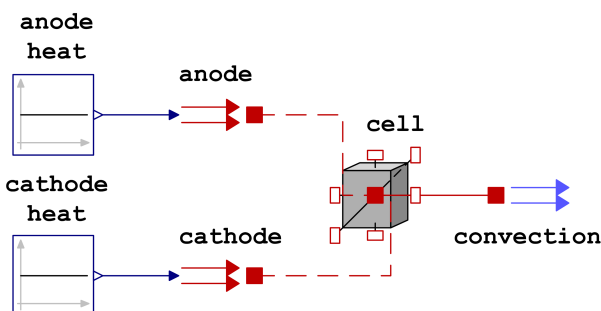


Figure 1: Thermal model of the cubical-shaped cell in *Modelica*

material etc. The heat losses of the anode and the cathode of the cell are implemented in the external models (anode and cathode in figure 1). The natural convection in this model is included. Each discrete volume, which is located on the surface of the modeled cell, contains heat transfer with the surrounding air. This heat transfer is identified as `convection`, figure 1.

Figure 2 shows the temperature distribution of a cubical-shaped cell in all three directions, x-length, y-length and z-length. The model considered 500 discrete volumes. Due to the origin of the losses in the area of anode and cathode the temperatures are higher in these discrete volumes.

The thermal model of the cell presented in this case simulates the thermal behavior of a cubical-shaped cell until the cell temperature reaches its stationary final value.

For the investigation of the thermal behaviour of the battery package which includes cylindrical cells, at first a detailed thermal model of one cell is implemented. This base model includes the same equations

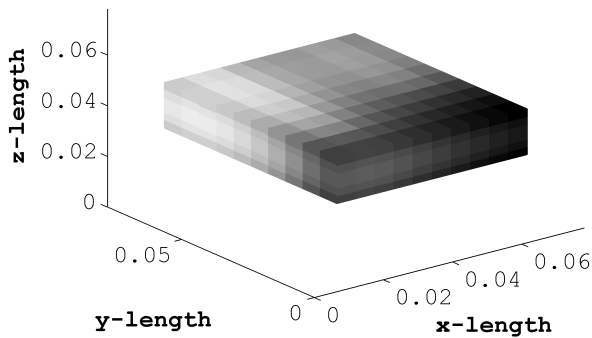


Figure 2: Simulation, temperature distribution in the cubical-shaped cell

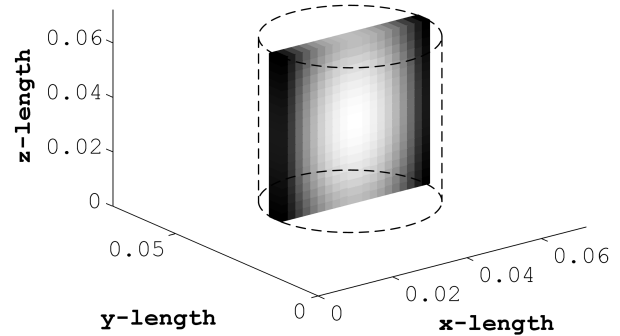


Figure 3: Simulation, temperature distribution in the cylindrical cell

and models as the cubical-shaped cell model. The anode and cathode of the cylindrical cell model are used in this case as spiral plates.

The heat losses distribution of the anode and cathode in this cell is implemented homogeneously. The heat flow in x and y direction, cross directions, has the same size and conductivity coefficients. Therefore the model of the cell is reduced from a $3D$ to a $2D$ problem.

This reduced model of the cylindrical cell is used in this paper for implementation of a battery package which contains more alike cells. Figure 3 shows the temperature distribution of a cylindrical cell in all three directions, x -length, y -length and z -length. The model considered 400 discrete volumes. The discrete volumes in the centre area have a higher temperature due to the improved losses heat from anode and cathode, because the heat losses of each volume of this cell has the same size. As depicted in figure 3, the heat flow from the center in y direction is much higher than in z direction. That occurs, because the heat conduction coefficients in x and y direction are much higher than that in the z direction.

2 Thermal model of a nickel metal hydrid battery package

The simple thermal cell model was used for the implementation and validation of the nickel metal hydrid battery package. This package contains 153 cells, which were integrated in a steel housing. The cells have a cylindrical shape in this case. The housing box has the following geometrical sizes, length $468mm$, width $108mm$ and height $65mm$.

Each component of the battery package such as cells and housing needs a set of thermal and geometrical parameters which have to be determined prior to the simulation. They are used of base from cell and housing material specifications, according to [2], [3], [4] and [5]. For the parameterisation of the final simulation model the parameters have been adjusted and corrected through measurements results.

The natural convection coefficient in this model is $4.5W/m^2 \cdot K$ and the ambient temperature $20^\circ C$. The size of this coefficient is used from quiet ambient air, according to [2]. The thermal conductivity coefficients of the battery package box housing in length, width and height direction are $254W/m \cdot K$, according to [3]. The thermal conductivity coefficients of the used cylindrical cells in cross direction are $40W/m \cdot K$ and in length direction $4W/m \cdot K$, according to [4].

The thermal losses of the entire battery package are determined through measured minimum and maximum temperatures of the entire battery surface. The battery package model was simulated until the simulated minimum and maximum temperatures were the same size as the measured temperatures. Then the heat loss of the entire battery package was determined to be $105.57W$. The heat losses of a cell determined using simulation is $0.69W$. The maximum simulated temperature of the battery package is $56^\circ C$.

The first model of the battery package was simulated using more than 2800 discrete volumes. The problem of this discretization of the battery package was that each volume has more than 50 equations. The entire battery model has therefore more than 140000 equations and, hence, the simulation of this model was impossible. The size of the model was reduced remov-

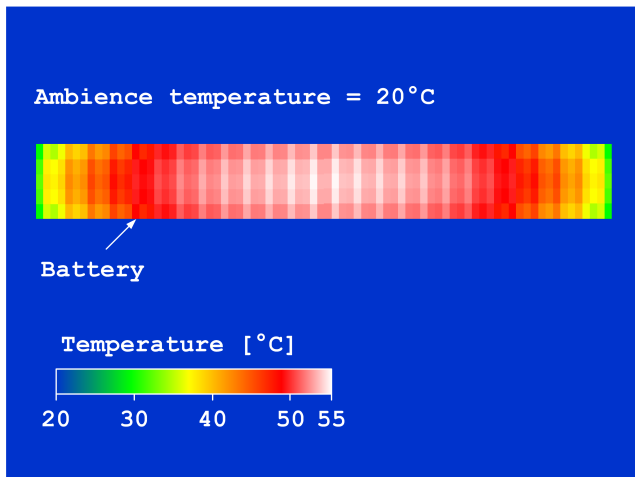


Figure 4: Simulation, temperature distribution in the battery package

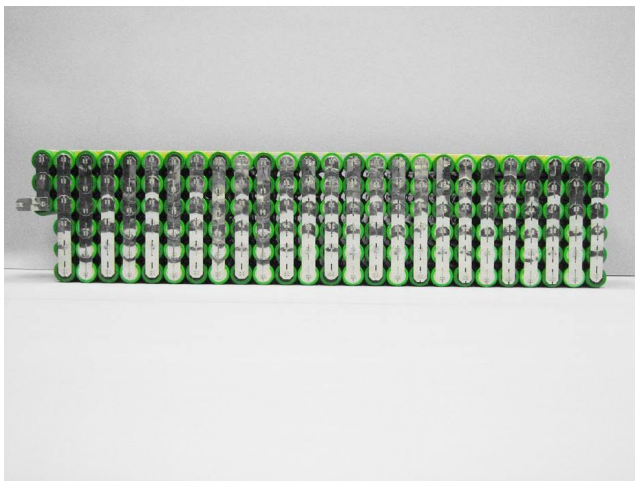


Figure 5: Assembly of the nickel metal hydrid battery package

ing surfaces where the heat flow rate was zero. With this optimization of the battery model the final model has less than 700 discrete volumes and therefore about 35000 equations.

Afterwards, simulation results were compared with measurement results of the real battery package. Figure 4 shows the temperature distribution of this simulated nickel metall hydrid battery package.

3 Test setup and testing of the nickel metal hydrid battery package

Thermal and electrical measurements have been carried out to validate the simulation results of both approaches of the thermal modelling. The nickel metal hydrid battery package consists of 153 single cells.

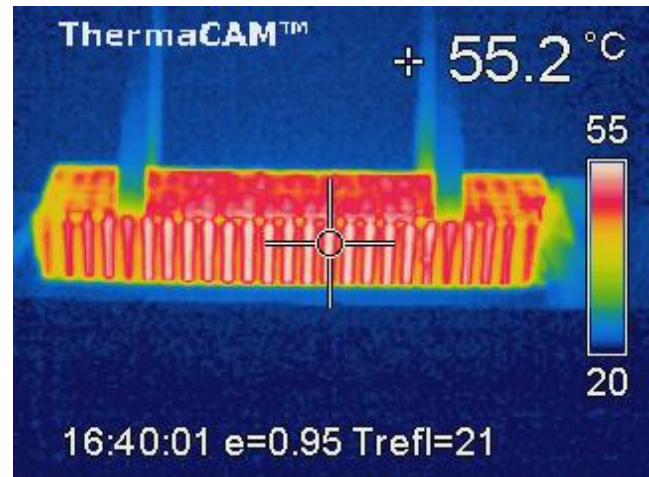


Figure 6: Thermographic picture of the nickel metal hydrid battery package

The cells are electrically connected in such a manner that the requirements for the given automotive application are fulfilled. In Figure 5 one can see the electrical and mechanical assembly of the nickel metal hydrid battery package.

The electrical internal resistance of a single cell depends on the state of charge of the cell, and the temperature of the cell.

The current profile for charging and discharging the the nickel metal hydrid battery package is a symmetrical current profile. Therefore the average state of charge of the cells stays constant. In this case the electrical internal resistance only depends on the cell temperature.

Several temperature sensors are applied in the battery package to control the temperature. The maximum temperature of a single cell must not exceed 60°C. The nickel metall hydrid battery is charged and discharged until the exponential temperature course reaches its stationary final value. The amplitude of the charge and discharge current is 9.66A. With this current profile the mean temperature of the battery package after reaching the stationary final value is 55°C.

During the heating-up of the battery pack a thermo-camera takes pictures of the pack to get the temperature distribution within the pack. Figure 6 shows a picture of the temperature distribution of the nickel metal hydrid battery package shortly before the mean temperature within the pack reaches its stationary final value of 55°C.

4 Electrical and thermal modelling of the nickel metal hydrid battery package

The stationary final temperature value of the battery modelled with this second approach will be the same as the stationary final temperature value simulated with the first model approach described in chapter 2.

The second model approach takes into account thermal and electrical components. The nickel metal hydrid battery package is modelled in *Dymola/Modelica* as a lump thermal mass with an internal heat source.

The heat supply is caused by the ohmic loss of the internal resistance of the nickel metal hydrid battery package, the temperature distribution is assumed to be homogeneous.

The internal resistance model is temperature dependent, consequently the generated ohmic loss depends on the temperature. Heat will be taken away from the nickel metall hydrid battery package due to natural convection.

The solved heat equation for a lump thermal mass with an internal heat source and heat exchange to the ambience is

$$\vartheta = \vartheta_a + \frac{P}{\alpha \cdot A} \cdot (1 - e^{-B \cdot t})$$

where

ϑ is the temperature of the nickel metal hydrid battery package,

ϑ_a is the ambient temperature,

α is the heattransfer coefficient,

A is the surface area of the package,

B is the heat up exponent and P is the heating power.

P is given as $P = R_i I^2$.

R_i is the temperature dependent internal resistance of the nickel metal hydrid battery package and I is the terminal current of the battery package.

The temperature dependence of R_i is approximated with a cubical function as one can see in Figure 7.

The *Dymola/Modelica* model of the nickel metall hydrid battery pack is charged and discharged until the exponential temperature course reaches its stationary final value. The simulated temperature course is almost identical with the measured temperature course of the nickel metall hydrid battery pack.

Figure 8 depicts the simulated temperature course with a red line and the measured temperature course with a blue line.

The stationary final value of the battery pack is 56°C which is identical with the simulated results of the thermal model presented in chapter 2.

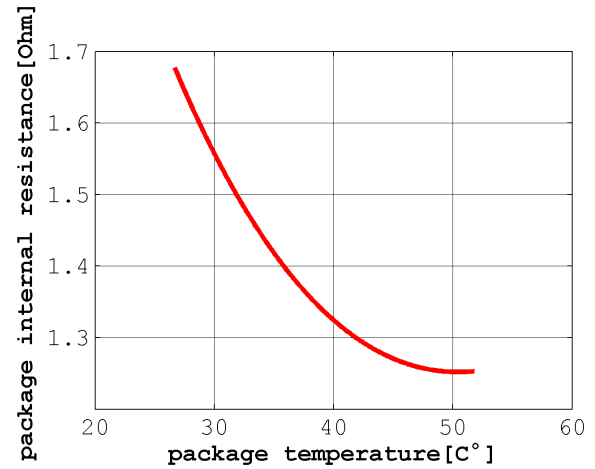


Figure 7: Temperature dependency of the internal package resistance, R_i

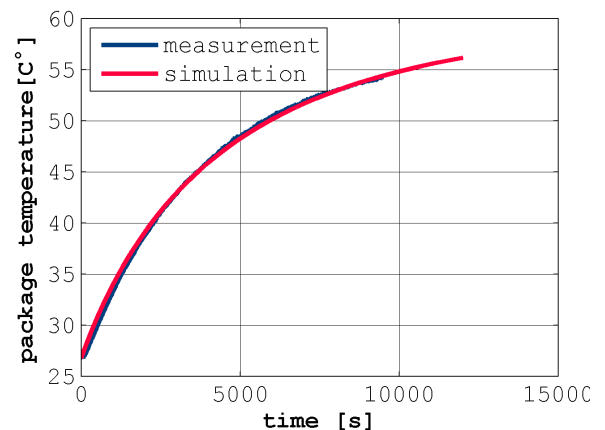


Figure 8: Simulated temperature course and measured temperature course of the nickel metall hydrid battery pack

5 Conclusions

In this contribution two thermal models of a nickel metall hydrid battery pack were presented. The results of the distinct approaches were the same, though.

The first model approach took into account the geometry and temperature distribution during the operation of the battery.

The second model approach took into account thermal and electrical components. Special emphasis was given to the temperature dependency of the electrical internal resistance of the battery pack.

Thermal and electrical measurements have been carried out to validate the simulation results of the thermal modelling and were presented.

The thermal models were implemented in *Modelica* simulation language and simulated using the *Dymola* simulation environment.

References

- [1] Peter Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press, Piscataway, NJ, 2004.
- [2] H.D. Baehr and K. Stephan, *Waerme- und Stoffuebertragung*, vol. 5. Auflage, Springer, 2006.
- [3] *Dubbel Interaktiv 2.0*, Verlag Springer electronic media, 2002.
- [4] VDI-Gesellschaft, *VDI-Wärmeatlas*, Springer, 2002.
- [5] R. Bosch, *Automotive Handbook*, Robert Bosch GmbH, Postfach 30 02 20, D-70442 Stuttgart, 5th edition, 2000.

Object Oriented Modeling of a Gasoline Direct Injection System

Matteo Corno Francesco Casella Sergio M. Savaresi Riccardo Scattolini
 Dipartimento di Elettronica e Informazione, Politecnico di Milano,
 Piazza Leonardo da Vinci, 32. 20133 Milano

Abstract

The topic of this paper is the object oriented modeling of a Common Rail Direct Injection System of a gasoline engine. The injection system of a gasoline engine is described; the main functional elements are detailed and an object oriented implementation using the Modelica language is proposed. The availability of a fast and easily reconfigurable simulator allows to study how different parts of the system interact and notably speeds up the design of the final system. The use of the Modelica language allows to seamlessly put together mechanics, fluid dynamics, and control algorithms. The design problem can be therefore approached as a whole, in a genuine and modern co-design approach.

Keywords: automotive; fluid dynamics; common rail injection system simulation.

1 Introduction

In this work we present an object oriented simulator of a Direct Common Rail Injection System of a gasoline engine.

The key to designing a clean and efficient ICE (Internal Combustion Engine) lies in precise control of the combustion. This can be achieved by accurate control of the flow of fuel and air in the combustion chambers. Pre-2000 injection systems (such as mechanical carburetors [10] and Multi Point Injection technology [9]) cannot meet today's stringent pollution regulations [8]. The introduction of the Common Rail Injection System technology for Diesel engines [1, 7] in the 90's represented a great breakthrough. Now, it is possible to precisely mix fuel and air directly in the combustion chamber. Only a few years had to pass before the same technology could be applied to gasoline engines [4, 5, 12], thus increasing fuel efficiency and reduce emissions. The cost of these advantages is a more complex system, both from the standpoint of mechanics and electronics. The higher complexity makes it

more difficult to foretell the effects of a modification of the elements of the system. The design of such a complex system can greatly benefit from the availability of a reconfigurable simulator. The design process can be sped up and the cost cut down.

The goal of this work is to describe an object oriented simulator of a modern Common Rail Injection System and to show how it can be helpful in the design of the injection system.

The work is structured as follows. Section 2 describes the overall architecture of the system. In Section 3, the mathematical model of the system is derived and its Modelica [2] implementation illustrated. In Section 4, it is shown how the model reconfigurability can be exploited for fast sensitivity studies. Finally, conclusions are drawn in Section 5.

2 Common Rail Injection System

A Common Rail Injection System schematics is depicted in Fig.1; the injection system goal is to deliver fuel to the injectors at a desired high pressure. The system can be divided into two sections; a high pressure circuit and a low pressure one. The low pressure circuit is composed of the fuel tank, a low pressure pump, filters and a pipeline. The low-pressure circuit is not critical for the overall engine performance and therefore it is out of the scope of this work. From the system dynamics standpoint, the most interesting part is the high-pressure circuit; it goes from the the high-pressure pump to the injectors. Its main elements are now briefly described:

- **HIGH PRESSURE PUMP.** It is a volumetric pump that connects the two main circuits of the system. It is used to increase gasoline pressure from 6 bar to [30-150] bar, according to the working load and engine speed. The piston of the pump is mechanically connected to the engine camshaft through a cam and follower system. In modern common rail pressure control system the control valve is built in the high pressure pump.

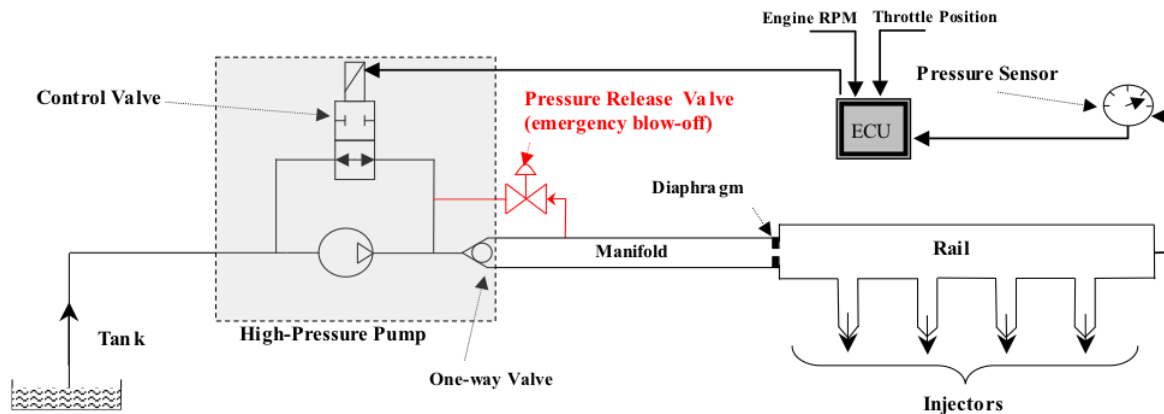


Figure 1: The Common Rail Injection System architecture.

Three main elements compose the pump: the piston, the control valve and the one-way valve. When the control valve is open, the output of the pump is redirected back to the low-pressure circuit. It has only two states: open or closed and the switching instant is the control variable of the system. The one-way valve is used to avoid unwanted refluxes.

Two phases are periodically alternated: an aspiration and a compression phase. In the first phase, the piston moves downward while the control valve is open. This allows the gasoline to flow from the low pressure circuit to the pump chamber. After the piston has reached its lower dead point, the compression phase starts. Initially the control valve is open and the gasoline flows back to the low pressure circuit. At any time during this phase the controller can command the closure of the control valve. When it is done, the pressure in the chamber increases. As soon as it surpasses the pressure in the high pressure circuit, the one way valve opens, letting the gasoline flow into the high pressure circuit. Notice that the control variable of the system is the closing instant of the control valve. The controller must be synchronized with the pump: when the piston is at its upper dead point (end of compression), the ECU (Electronic Control Unit) samples the pressure sensor output and computes the control action. The control action will be executed, at the earliest, when the piston reaches its lower dead point.

- **MANIFOLD.** The outlet flange of the high-pressure pump is connected to the common rail through the manifold. The one-way valve pre-

vents the gasoline from flowing back from the manifold to the pump. The manifold is equipped with a safety valve that opens whenever the pressure in the manifold reaches a threshold. This valve is designed to function as a safety device and its opening should be avoided in nominal conditions.

- **DIAPHRAGM.** The common rail and the manifold are connected through a diaphragm that reduces the diameter of the pipeline. This component provides for a better damping of the system and a partial decoupling between the pressures of the manifold and the pressure of the rail. This is achieved at the price of a decreased energy efficiency of the whole system.
- **COMMON RAIL.** It is the core of the system. It is connected to the manifold through the diaphragm. All the injectors are connected to the common rail. In order to achieve precise injection control, the pressure in the common rail must be regulated at the desired reference value, minimizing oscillations.
- **PRESSURE SENSOR.** It measures the pressure in the common rail at its end. It is the only pressure measure available for control.

3 Object Oriented Modeling

In this section the mathematical modeling of the system and its implementation in the Modelica language are described. First, the core of a simplified fluid dynamics library is described and then it is shown how it is possible to use the library (along with the Modelica Standard Library) to implement the injection sys-

tem. Modelica turns out to be well suited for aiding the design of injection systems. Injection systems are inherently multi-domain systems and the possibility to change each element of the system without having to redesign all the interconnections allows to easily study the effects of different design choices. The use of such a fast and user-friendly virtual prototyping system can save numerous iterations of the design-prototyping-testing cycle.

3.1 Simplified Fluid Dynamics Library

Being fluid dynamics the most important dynamic phenomenon of the system, a new simplified fluid dynamics library has been developed. The library is built on the two new connectors: the `flange_in` and `flange_out` defined as:

```
connector flange_in
  "Connector for the simplified
  fluid dynamics library"

  import Modelica.SIunits.AbsolutePressure;
  import Gasoline_turbo.Types.MassFlow;

  flow MassFlow q
    "Mass flow into the flange";
  AbsolutePressure p
    "Pressure at the flange interface";

end flange_in;
```

The absence of a temperature variable in the connector deserves a comment; the library is not designed to model thermal interaction. This choice is based on the consideration that, although thermal interaction does take place in the system, the focus of this study is on pressure waves. Wave propagation is faster than any thermal interaction that may happen in the system and therefore heat exchanges can be neglected.

In addition to the connectors, two partial packages have been introduced: the `BaseFluid` and `BaseMaterial`. The former representing the fluid properties and the second representing the properties of the material of the pipeline.

```
partial package BaseFluid

import Modelica.SIunits.AbsolutePressure;
import Modelica.SIunits.BulkModulus;
import Modelica.SIunits.KinematicViscosity;
import Modelica.SIunits.Density;
```

```
replaceable partial function getDensity
  "Return density as function
  of absolute pressure"
  extends Modelica.Icons.Function;
  input AbsolutePressure p "Pressure";
  output Density rho;
end getDensity;
```

```
replaceable partial function getBulk
  "Return BulkModulus as function
  of absolute pressure"
  extends Modelica.Icons.Function;
  input AbsolutePressure p "Pressure";
  output BulkModulus beta;
end getBulk;
```

```
replaceable partial function getViscosity
  "Return Viscosity"
  output KinematicViscosity vcin;
end getViscosity;
```

```
end BaseFluid;
```

The main properties used in the modeling are density, bulk modulus and kinematic viscosity. The package allows to add new properties where needed and redefine how the properties are computed. For example, the designer can choose to use a constant value for the density or model it as a function of the pressure. This gives the user a certain amount of freedom whenever the fluid properties are not readily available. In this injection system study a gasoline package has been implemented assuming a linear dependence of the bulk modulus on the pressure.

Similarly the `BaseMaterial` package is defined as:

```
partial package BaseMaterial

import Gasoline_turbo.Types.*;

replaceable partial function getYoung
  extends Modelica.Icons.Function;
  output Young y;
end getYoung;

replaceable partial function getPoisson
  extends Modelica.Icons.Function;
  output Poisson p;
end getPoisson;

end BaseMaterial;
end BaseFluid;
```

The main properties of the material are assumed to be the Young and Poisson moduli. They characterize the material elasticity, which plays an important role in pressure waves propagation. In the final implementation, the two moduli are assumed constant, but the replaceable feature allows to take into account more complex models. The partial package has been extended into two different types of steel: the manifold steel and the common rail steel.

3.2 Models Description

As already mentioned the most important dynamics affecting the system is the propagation of pressure waves in narrow, long, circular section pipelines filled with a compressible fluid. According to the description given in Section 2, the injection system is obtained by connecting the following elements: pipelines, diaphragm, high pressure pump, and valves. The closed-loop pressure control and synchronization algorithms also need to be modeled.

Fig. 2 is a graphical representation of the Common Rail Injection System. The most important elements of the system are depicted in figure and are described in the following.

3.2.1 Common Rail and Manifold

The DistributedPipe is constituted by two connectors, a `flange_in` and a `flange_out` and the two replaceable packages, `BaseFluid` and `BaseMaterial`; this allows to specify the fluid and material properties independently of the wave dynamics model.

The distributed pipe model equations are derived under the following assumptions:

1. single component, single phase fluid;
2. one-dimensional spatial model;
3. negligible heat exchange phenomena;
4. straight and constant section pipelines.

Under these assumptions, the dynamics of the fluid in the pipeline are described by mass and momentum balances, which can be written as [11]:

$$\begin{cases} \frac{A}{c^2} \frac{\partial P}{\partial t} + \frac{\partial w}{\partial x} = 0 \\ \frac{\partial}{\partial t} \left(\frac{w}{A} \right) + \frac{\partial}{\partial x} \left(\frac{w^2}{A^2 \rho} \right) + \frac{\partial P}{\partial x} + F_f = 0 \end{cases} \quad (1)$$

In system (1), ρ is the gasoline density expressed in kg/m^3 ; w is the gasoline mass flow expressed in kg/s ; P is the gasoline pressure expressed in Pa; c is the sound

velocity in the fluid expressed in m/s ; A is the section area of the pipe expressed in m^2 ; and F_f is the load loss due to friction. The sound velocity term c depends on the properties of the fluid and the elasticity of the pipe and it is given by:

$$c = \sqrt{\frac{\beta}{1 + K\beta} \frac{1}{\rho}} \quad (2)$$

where β is the bulk modulus that describes the compressibility of the fluid and K is the stiffness of the pipe that depends on its material and on its geometric properties. In addition, assuming turbulent flow (this assumption is verified a-posteriori) the frictional load loss can be written as:

$$F_f = 2 \frac{w|w|}{\rho A^2 D} f \quad (3)$$

where D is the inner diameter of the pipe and f is the pipe Fanning friction factor. The final equations are partial differential equations. The infinite-dimensional system can be transformed in a finite-dimensional one by means of the finite-difference method [3, 6]. The pipeline is divided in N cells, each assumed to have a uniform pressure and an inflow and outflow. The finite-difference approximations used herein are (the dependence of P and w on t is omitted for the sake of notational simplicity) :

$$\begin{aligned} \frac{\partial P}{\partial x} &\approx \frac{P(i+1) - P(i)}{\Delta x} \\ \frac{\partial w}{\partial x} &\approx \frac{w(i) - w(i-1)}{\Delta x} \end{aligned} \quad \text{for } i = 2 \dots N \quad (4)$$

In (4), Δx represents the cell length; P is the mean pressure within the i -th cell; and $w(i)$ and $w(i-1)$ are the inlet and the outlet flows of the i -th cell, respectively, and N is the number of cells considered. The discretization must be completed by boundary conditions; this is done using the `flange_in` and `flange_out` connectors already introduced:

$$\begin{aligned} w(1) &= \text{flange_in.q} \\ w(N+1) &= -\text{flange_out.q} \\ p(1) &= \text{flange_in.p} \\ p(N+1) &= \text{flange_out.q} \end{aligned} \quad (5)$$

These boundary conditions allow to freely connect different elements without having to worry about causality of the elements, as one would have to do in a signal based modeling environment.

The distributed pipe is used to model both the manifold and the common rail. The manifold is simply modeled by a long distributed pipe; it is connected to

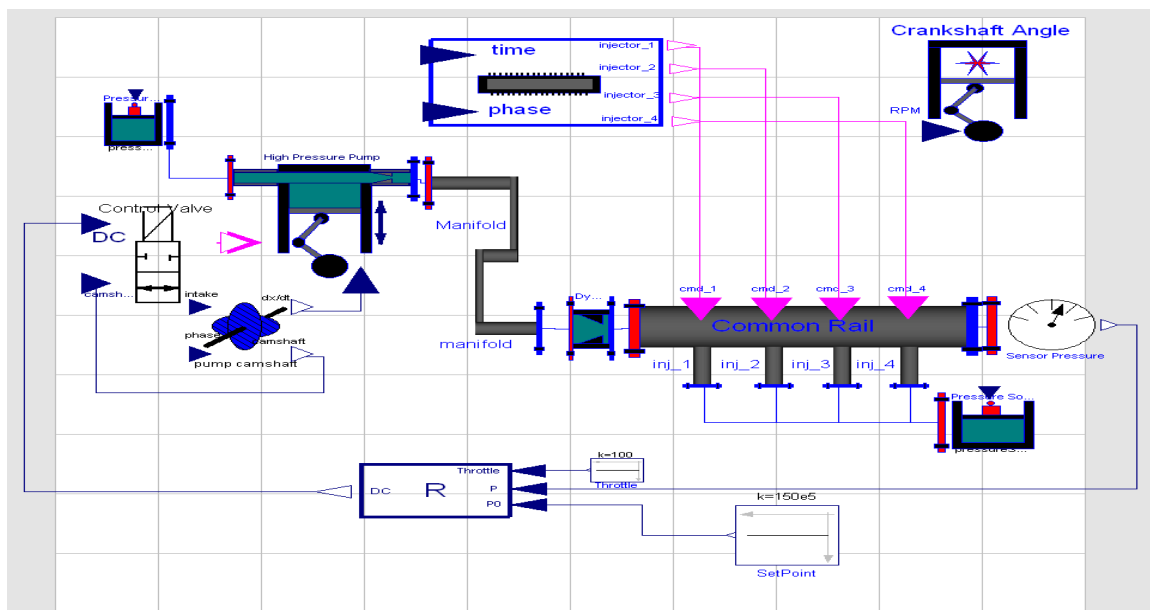


Figure 2: A graphical representation of the Injection System Model.

the high pressure pump on one side and to the orifice on the other. The modeling of the common rail is more interesting because it must be provided with the connectors used by the injectors. Fig. 3 is a graphical representation of the CommonRail model. It shows the model interface and its components. The

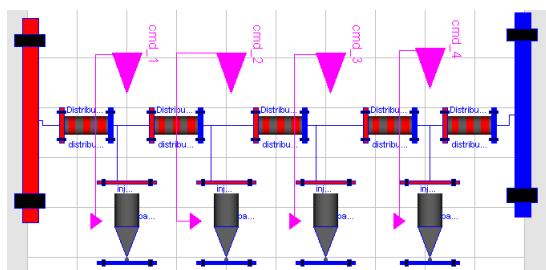


Figure 3: A graphical representation of common rail model.

model is composed of 5 distributed pipes and 4 injectors. The interface is represented by the two main flanges of the common rail, the 4 flanges of the injectors (which will be described later) and 4 logical signals representing the command signals to the injectors. The model encapsulates the geometric properties of the rail (length, diameter and inclination) and the two replaceable packages Basefluid and BaseMaterial. As shown in Fig. 1, in the final assembly, one of the main flanges of the common rail is connected to the diaphragm, the other to the pressure sensor. The injectors flanges are connected to a constant pressure representing the piston chambers.

3.2.2 Diaphragm

The diaphragm is a choking of the pipe. Its function is to increase the damping and by doing so it also achieves a certain amount of decoupling between the manifold and the rail dynamics. The narrowing is coaxial to the fluid flow; thus it can be modeled as concentrated load loss [11]. Having defined $\Delta p = P_1 - P_2$, the flow characteristic can be written as

$$w = \begin{cases} \rho \pi \frac{D_i^2}{4} \sqrt{\frac{2|\Delta p|/\rho}{1-(D_i/D_o)^4}} & \text{if } \Delta p > 0 \\ -\rho \pi \frac{D_i^2}{4} \sqrt{\frac{2|\Delta p|/\rho}{1-(D_i/D_o)^4}} & \text{if } \Delta p \leq 0 \end{cases} \quad (6)$$

where D_i is the diameter of the inlet and D_o the diameter of the choking.

The resulting model has two flanges; one is connected to the manifold and one to the common rail. It is important to notice that this element must manage flow inversion. Flow inversion happens when the pressure gradient between the two flanges changes sign. From equation (6), it is immediate to see that when ΔP approaches 0 the Jacobian of the function approaches singularity. In order to treat this case a linear junction has been implemented. When $|\Delta P| < \epsilon$, the flow characteristic is approximated by a linear function.

3.2.3 Injectors

The injectors are one of the elements of the system most likely to be subject of study; in order to increase the configurability of the model, the partial model BaseInjector has been implemented. Its interface

is composed of two flanges and of a boolean signal. In a standard configuration, one of the flanges is connected to the common rail and the other to the cylinder chamber.

For this study a simple injector model has been implemented. It is modeled as a valve whose opening request is assumed to be a two-valued variable (open-close). The dynamics of the orifice area is approximated by a first order filter with a pure delay. The relationship between the actual orifice area and the output fuel flow is assumed to be a non-dynamic relationship and modeled by a static customizable map depending on the pressure difference between the two flanges. Notice that the cylinder chamber can be modeled as a constant pressure. Although the pressure in the chamber changes during a piston revolution; the high pressure in the common rail guarantees that the flow through the injectors is independent from the chamber pressure. It is also interesting to note that the injectors are not subject to flow inversion and thus their implementation is straightforward.

3.2.4 Engine Carrier

In engine control applications, it is a common practice to write all the control logic algorithms in term of the engine crankshaft angle. In order to meet this standard, a mechanism to relate the time independent variable (needed to simulate wave propagation) and the engine crankshaft angle independent variable (needed to simulate injection, spark, pumping and control) is necessary. The `EngineCarrier` model achieves this goal. The crankshaft is computed as a function of the instantaneous RPM of the motor (an user supplied variable). By defining an inner `EngineCarrier` instance in the model, the user can conveniently provide all the models with the needed independent variable.

3.2.5 High Pressure Pump and Control Valve

The high-pressure pump is one of the most important elements of the system; and, according to the object oriented paradigm, it has been defined as a partial model `BasePump`. The partial model only defines its interface (2 flanges, a command signal, and a real signal describing the velocity of the piston) and its main components (the replaceable packages `fluid` and `material`). This approach allows to easily define and implement different models of the pump.

For the goals of this study the compression dynamics inside the pump chamber is neglected along with the dynamics of the low pressure circuit. This is done be-

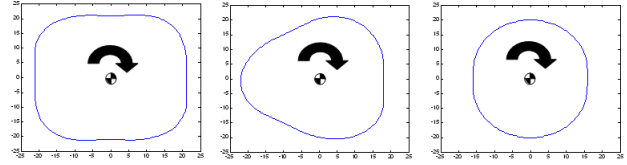


Figure 4: High pressure pump cam profiles (4-lobes, 3-lobes, 2-lobes).

cause the chamber volume is negligible with respect to the volume of the rest of the system. As described in Section 2, the piston is driven by a camshaft and the instantaneous gasoline flow is determined by the velocity of the piston.

$$w_h = 0, \quad w_l = -\rho A_{pist} \frac{dh}{dt}, \quad \text{if control = open}$$

$$w_h = \rho A_{pist} \frac{dh}{dt}, \quad w_l = 0, \quad \text{if control = closed}$$

where w_l and w_h are, respectively, the flow through the flange connected to the low and high pressure circuit; A_{pist} is the area of the piston; h is the height of the piston and control is the state of the control valve.

The other two models needed in order to have a functional high pressure pump are the `PumpCamshaft` and the `ControlValve` models. These two models require the crankshaft angle provided by an `EngineCarrier` instance. `PumpCamshaft` computes the velocity of the piston as a function of the crankshaft angle; this is important because different high pressure pump camshaft profiles may be available, and the ability to easily switch between them is helpful. Three different cam profiles have been implemented: 2, 3, and 4 lobes. They are depicted in Fig. 4. The `ControlValve` model determines the state of the control valve. Its interface is defined by 2 input signals (DC and camshaft angle) and one boolean output signal which represents the state of the valve. As explained in Section 3.2.7 the DC signal is the control variable; it is the closing time of the valve expressed as a ratio to the whole high pressure pump period. The model can be extended to define more complex behaviors.

3.2.6 Injection logic

The `InjectorLogic` model generates the command signals for the injectors. It is possible to specify a time variant or constant injection time, in seconds, and an injection phase, in crankshaft degrees. It is therefore possible to simulate engine steady state or transient regimes. It requires the presence of an inner

EngineCarrier instance in the model. In the present implementation, which is based on real-world specifications, the four injection times are equal and sequentially delayed by 180° .

3.2.7 Control Algorithm

The goal of the common rail pressure controller is to regulate the pressure to a set point which is computed as a function of engine rpm and the throttle position. A known and steady pressure in the common rail allows to control the injected gasoline with precision by action of the injection time. This can be explained by the above considerations regarding the injectors; thanks to the great pressure difference in the rail and in the piston chambers, the amount of injected fuel depends mainly on the pressure in the rail and the injection time. If the pressure in the common rail is maintained constant without oscillations, then the injected fuel can be controlled by varying the injection time. Regulation of the gasoline pressure in the common rail is achieved in closed-loop, with an additional feed-forward component. The only measurement available for feedback is provided by the rail pressure sensor whose reading is sampled when the high pressure pump piston reaches its upper dead point (start of the expansion phase). The sampling time of this sensor hence is time-varying (whereas it is camshaft-angle-invariant). The control variable is the closing time of the valve measured as a Duty Cycle. The Duty Cycle (DC) is referred to the whole high pressure pump cycle, starting with the aspiration phase; therefore a DC of 50% means that the control variable is closed as soon as the compression phase begins, whereas a DC of 75% implies that the control valve is closed in the middle of the compression phase. Hence, this actuator implements a sort of Pulse-Width-Modulation (PWM). It is clear that, according to this control strategy, the aspiration phase is a pure delay. Also in this case, notice that the duty-cycle is time-varying but camshaft-angle-invariant. The control algorithm is implemented in the Controller model. It has 3 inputs (pressure measurement, pressure set point and throttle position) and one output, the Duty Cycle, an Controller instance provides the variable for scheduling. Fig. 5 depicts the control system block diagram. The control algorithm is based on a Proportional + Integral control action whose parameters are scheduled on the engine speed. The feed-forward component allows to compensate for the injected fuel. The injected fuel can be seen as a load disturbance acting on the plant; although, technically, this disturbance cannot be

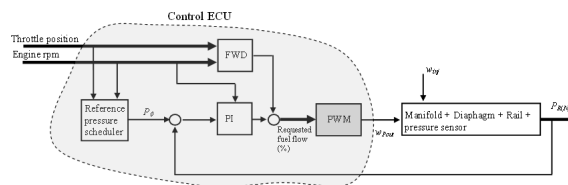


Figure 5: Control system block diagram.

measured, it can be estimated by knowing the gasoline pressure in the common rail and the injection time. The feed-forward term guarantees a faster disturbance rejection than the one achievable only by the closed loop term.

4 A Case Study

The implemented model can be used for different kinds of sensitivity analysis. In this section the simulation results are presented and commented. Before doing any sensitivity analysis, the fluid parameters have been identified and the model validated. The model is characterized by a large number of parameters. They belong to two categories: geometric and fluid-dynamic parameters. Geometric parameters are easily known; fluid dynamic parameters cannot be measured easily and inexpensively and therefore they were experimentally estimated. The parameters that have been estimated from data are: the gasoline bulk modulus and the Fanning friction factor of the manifold and of the rail. Note that, although the bulk modulus of the gasoline is known [11], it must be corrected to account for the elasticity of the pipes which is hard to be directly measured. All model uncertainties and simplifications are concentrated in the identified parameters. Fig. 6 shows the final validation results, by plotting the manifold and rail pressures. Pressures are normalized with respect to the set point pressure. The validation data is the result of a workbench test on a modern turbocharged direct ignition gasoline engine. It is important to note that only closed loop tests are available for validation; therefore all the following considerations are to be referred to the closed loop case. The results show that the model is able to accurately replicate the main resonances and damping. The relevant dynamics are correctly captured. Spectral analysis confirms that the frequency range of validity of the model is approximately 0-1000Hz. It can be seen from figure that the model is able to reproduce higher frequency dynamics, but the fitting between the simulated and the measured data is not as

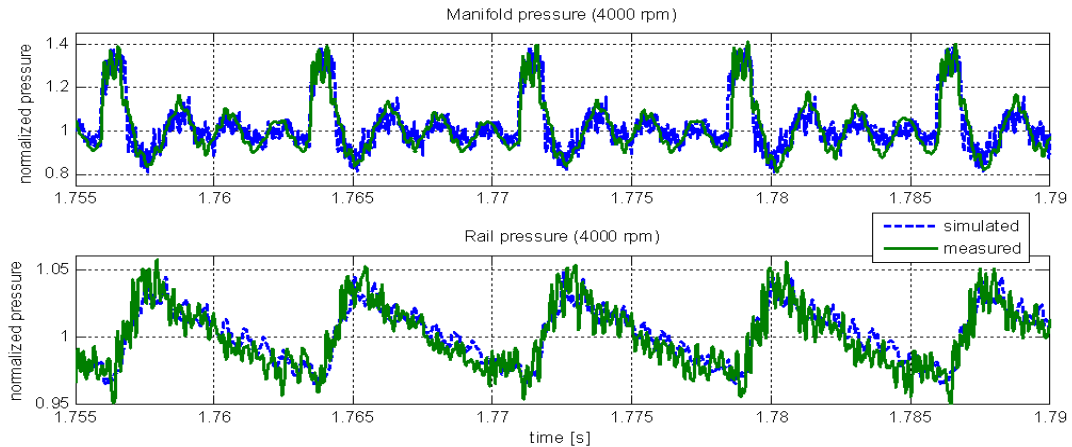


Figure 6: Simulated and measured pressures in the manifold and in the rail (time-domain; 4000rpm).

good as with lower frequency dynamics. In order to improve the model accuracy beyond 1KHz, the Fanning friction coefficient dependence on the Reynolds number needs to be accounted for. Although the identification of such dependence requires considerable efforts; the model can be easily modified to account for that dependence.

This validation test also shows that the prototype suffers from high pressure peaks. In fact, the pressure in the manifold raises to levels up to 1.4 times the nominal pressure. The final system will be equipped with an exhaust valve set to open at about 1.35 time the nominal pressure. The opening of the safety valve is to be avoided.

This problem is well suited to be studied with the described model. The dependence of the pressure peaks has been studied as a function of the diameter of the manifold. Results are shown in Fig. 7, where pressures are normalized with respect to the safety valve threshold. Thanks to this analysis, it is possible to draw some interesting guidelines for the design of the manifold:

- the pressure peak problem is sensitive to the engine speed. The higher the engine speed is the higher the pressure peak is;
- the maximum pressure depends hyperbolically on the diameter of the manifold;
- an increase of the manifold diameter of 2mm with respect to the baseline manifold solves the problem.

This is an example of the kind of possible sensitivity analyses that can be run using the model. Other elements that can be easily studied are: the radius of the rail, the number of lobes of the high pressure pump and the static characteristic of the injectors.

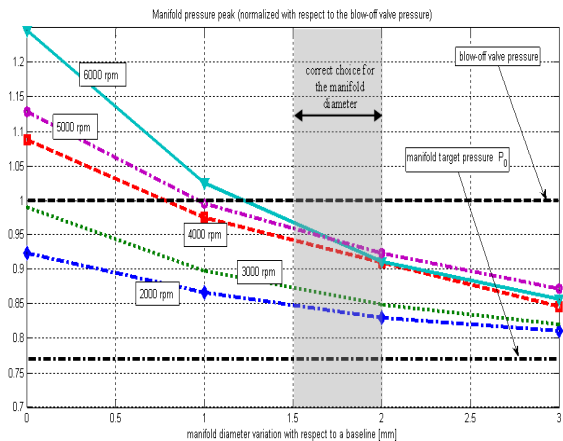


Figure 7: Manifold maximum pressure as a function of the manifold diameter increment for different engine speeds.

5 Conclusions

In this work an object oriented model of a Common Rail Injection System for a gasoline engine has been proposed. The focus has been on providing an easily reconfigurable simulation tool that can help the design process. The system has been described in detail and a 1-Dimensional model of the pressure waves propagation has been derived from fluid dynamics basic principles. The simulator has been validated using bench tests. A case of study in which the model is used to study the dependence of the maximum pressures reached in the manifold as a function of the manifold diameter has been presented. The case of study shows that the proposed model is accurate enough to help co-design of the system, where mechanics, fluid dynamics and control logic are looked at as an ensemble. These kind of analyses allowed to draw useful

guidelines for the design of such a complex system.

6 Acknowledgments

The Authors would like to thank Emilio Comignaghi, Marco Sofia (from FIAT Powertrain srl), Antonio Palma and Eduardo Sepe (from Élasis SCpa) for the insightful discussions and valuable suggestions.

References

- [1] Kouremenos D. A. and Hountalas D. T. Development and validation of a detailed fuel injection system simulation model for diesel engines. *SAE Technical Papers*, 01(0527), 1999.
- [2] Modelica Association. Modelica - a unified object-oriented language for physical system modelling. language specification. technical report, 2002.
- [3] Bertin D., S. Bittanti, and S.M. Savaresi. The decoupled cushion control in ride control systems for air cushion catamarans. *Control Engineering Practice*, 08, 2000.
- [4] Alabastri E., L. Magni, S. Ozioso, R. Scattolini, C. Siviero, and A. Zambelli. Modeling, analysis and simulation of a gasoline direct injection system. In *Proceedings of the IFirst IFAC Symposium on Advances in Automotive Control*, 2004.
- [5] D'Errico G. and Onorati A. An integrated simulation model for the prediction of gdi engine cylinder emissions and exhaust after-treatment system performance. In *SAE Int. Congress & Exp. Detroit*, 2004.
- [6] Strikwerda J. *Finite Difference Schemes and PDE*. Wadsworth Brooks/Cole., 1980.
- [7] Chiavola O. and Giulianelli P. Modeling and simulation of common rail systems. *SAE Technical Papers*, 01(199), 1999.
- [8] Official Journal L 044, 16/02/2000. *Directive 1999/96/EC of the European Parliament and of the Council*.
- [9] Baker P. and Watson H. Mpi air/fuel mixing gaseous and liquid plg. *SAE Technical Papers*, 01(246), 2005.
- [10] Covington J. P. Modernizing the fixed-venturi carburetor. *Automotive Engineering*, 82(7), 1974.
- [11] Thomas P. *Simulation of Industrial Processes for Control Engineers*. Butterworth Heinemann, Oxford, England., 1999.
- [12] Li J. Z., Treusch C., Honel B., and Neyrat S. Simulation of pressure pulsations in a gasoline injection system and development of an effective camping technology. *SAE Technical Papers*, 01(1149), 2005.

Session 1d

Electric Systems & Applications

A Multi Level Approach for Aircraft Electrical Systems Design

† Martin R. Kuhn † Martin Otter ‡ Loïc Raulin

† German Aerospace Center (DLR)
Institute of Robotics and Mechatronics
Department of System dynamics and Control
82234 Wessling, Germany

‡ Airbus France
Electrics System
EDYNE department
31300 Toulouse, France

Abstract

This paper describes the needs, ideas, implementation and application of a multi-level concept used for aircraft electrical systems design. The goal is to easily switch between three model levels in a complex system model, in order to arrive at dedicated models for the needed simulation tasks: a simple and super fast model for energy consumption design, a detailed model for fast network stability analysis and a very detailed model for network quality assessment. Special care was spent on the modeling assumptions and a suitable library concept fitting to the needs. For simplified unitary testing and configuration management of the multi-level models a concept was developed. The approach is demonstrated with an aviation equipment use case. The usage of the models for stability and quality studies is sketched.

Keywords: Multi-level modeling, Electrical Network, Aircraft, DC-DC buck converter, stability analysis, quality analysis

1 Introduction to multi-level modeling

For industrial design, evaluation and certification process of energy distribution networks, often several models exist which represent different modeling accuracies of the same system. It is not always useful to take the most detailed one, as it may not improve the overall accuracy but will slow down the simulation drastically. Depending on the desired evaluation of an electrical network (power consumption, network stability, network quality), system simulations with models of different levels of accuracy are much better

suited. The simulation time of the simplest, architectural level of a model is usually 2 to 3 orders of magnitude faster than the most complicated, behavioral level of the model.

The following model levels are taken into account:

- Level 1: Architectural level
Steady-state power consumption. Usually, algebraic equations describing the energy balance between ports without dynamic response.
Typical use: power budget.
- Level 2: Functional level
Steady-state power consumption and mean-value transient behavior (e.g. inrush current, consumption dynamics with regard to input voltage transients). Switching is not included.
Typical use: network logic studies, network stability studies.
- Level 3: Behavioral level
Representing actual wave forms including switching and HF injection behavior.
Typical use: network power quality studies.

Note: This nomenclature may be used differently in the literature.

In Figure 1 on page 2 the tree-level concept is illustrated at hand of simulations of a DC/DC buck converter. The architectural layer input current shows large simulation steps neglecting the detailed effects which can be seen at the behavioral layer model simulation. The functional layer model covers the waveform of the detailed model without switching effects. In order to improve the simulation process for design and validation of the Electrical System, there was

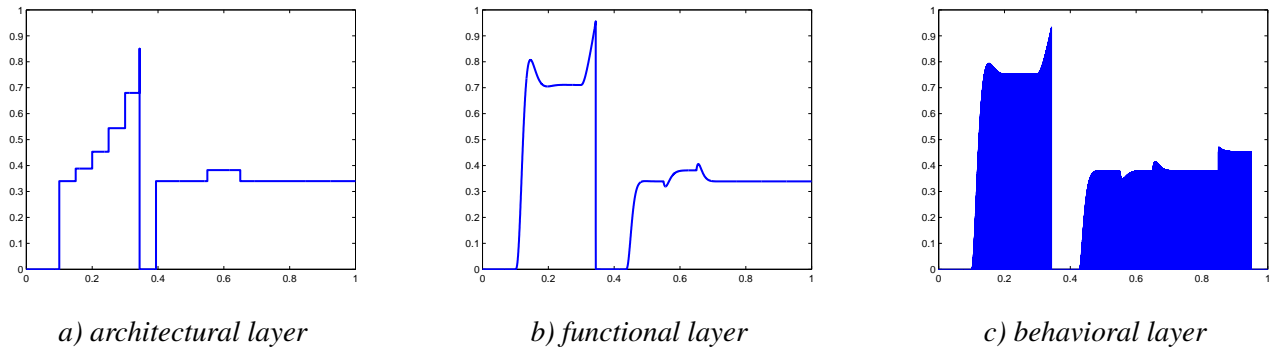


Figure 1: Illustration of the 3 layer concept at hand of simulations of a DC/DC buck converter

identified a need for methods to simplify unitary testing and configuration management of suppliers' models in order to enhance the coherency of electrical behavior of the various modeling levels of one single equipment.

2 Modeling assumptions

While the architectural layer is defined to contain only static energy balance equations and the behavioral models are close to the hardware level, the definition of the functional models is not as straight forward. These models have to be applicable for linearization for stability investigations with methods of control theory for linear time invariant systems. For DC systems, those have to be deduced from the complex models by averaging the switching models on a time interval and order reduction techniques.

For AC systems, the time variant characteristics of the sinusoidal alternating currents prevents a time invariant steady state condition and hence linearization. Therefore it is proposed to use an equivalent representation which expresses a rotating electric machine in a static rotor fixed system (also called Park transformation)¹. Transforming the system equations to a net-frequency fixed system results e.g., in " $i(t) = i(\omega_{net}) + \Delta i(t)$ ". The model uses $\Delta i(t)$ as state and not $i(t)$. In stationary operation with constant load and constant speed, $\Delta i(t) = 0$ and $i(t) = i(\omega)$. This time invariant system also results in much faster simulation. The transformation output is a representation in d, q and zero system. The zero system may be skipped for this application since it is mainly relevant for asymmetric loads not treated by functional models. Park trans-

formation is the standard way of modeling generators and motors and used for control but is also applicable to 3phase-line impedances. Asymmetric loads are allowed but will not result in a steady state condition for this transformation. For an aeroplane multi-phase power transmission system the following components are of importance:

- Power source: The generators supply sinusoidal voltage at net frequency. Calculation of voltage and current is usually performed in a rotor-fixed system (synchronous machine)
- Loads: For symmetrical resistive, inductive and capacitive loads as well as synchronous motors current/voltage relations are fixed to net frequency.
- Other loads: Switches, diodes, unsymmetrical loads, time variant loads do not have simple steady-state dependencies upon net frequency. But: Calculation of the generator demands the use of transformations anyway. So, incorporating other components to the dq0-system will not necessarily essentially speed up the simulation but will not break it either.

Limitations for level-triggered switching devices may be circumvented by averaged models. Especially for self commutating rectifiers there is a need for calculating the mean output voltage/current, averaged on a commutation interval. Theory on electric components, the transformation and averaged models can be found for instance in [1].

¹Some results were inspired by the EU Project Realsim (Real-time Simulation for Design of Multi-Physics Systems). It resulted in the open source library "SPOT" of H. J. Wiesmann of ABB Switzerland, <http://www.modelica.org/libraries/spot>

3 Library concept

3.1 Comparison of methods

Following the demands from chapter 1, methods for multi-level modeling were investigated which could replace the current models (behavioral, functional, architectural) by one single multi-level model. This multi-level model shall integrate the separate layers to be activated independently from each other. Therefore a methodology is investigated to join models of different abstraction levels within a single "container" model and supply tools for the automatic selection of the desired level. Three alternatives were identified to meet the demands:

Alternative 1: Every level is implemented as a separate sub-model. A "generic" model (or template model) is utilized in a user model via the Modelica language construct "replaceable". Via a selection box the "generic" model can be defined to be one of the level models. The connectors of the different levels are either identical or, if this is not possible, they are also "replaceable". Details may be found in the Modelica tutorials. Typical problems are:

- It is not possible to select the level by an expression, i.e. it is not possible to select the model based on the setting of the global default. Instead, at every component the model level has to be manually changed to the required one.
- Whenever the selection is changed (e.g. from architectural to behavioral and then back to architectural), all parameter definitions from the previous selection are lost.
- All levels need the same connectors or the connectors must be replaceable, which makes the design complicated.

Alternative 2: All layers are described in the same model. Via flags different parts of the model are activated. In the simplest case, there is just an if-clause:

```
parameter Integer level(min=1, max=3) = 1 "Model
level";
...
equation
  if (level == 1) then
    // equations for architectural model
  elseif (level == 2) then
    // equations for functional model
  else
    // equations for behavioral model
  end if;
```

The simulation program selects at compile time the corresponding if-branch, if the branch is determined by a parameter expression (i.e., in the generated code, only the equations of the selected branch is present).

Advantages:

- The level can be set by an Integer expression. This allows, e.g., to refer to a global setting of the level.
- The parameters of all 3 levels are visible and can be set. They remain present, even if the level is changed. This alternative is useful for base components, such as a capacitor or an inductor, that are described by equations and do not contain other model instances.

Alternative 3: This is a variant of alternative 2. Every level is a separate model. There is a "container" model that is used in a user model. The container model contains all models of the different levels in form of conditional models. Via a flag, the desired model level is activated and the connect statements to the deactivated submodels are removed automatically. Advantages:

- Every model level can be built and tested independently from the other levels. Only the connection interfaces need to be the same.
- When switching between levels, the parameter settings of the other levels remain.
- Parameters that are common to all levels can be defined in the container object and can then be propagated to the level models (via Modelica modifications)

Disadvantages:

- In the plot window there is an unnecessary hierarchy, e.g., motor.level2.flange_a.w (on the other hand, it becomes very clear which level is contained in the model).
- The connection lines from the level models to the interfaces are redundant information (e.g. not present in alternative 1)

Due to severe disadvantages with implementing this so called "conditional declaration" with existing language constructs, an initiative was started in the Modelica Association to improve this situation. As a result, in the Modelica language version 2.2 from Feb. 2005,

“conditional declarations” have been introduced into the Modelica language. This language construct has been supported in Dymola since March 2005 (Dymola version 5.3c).

3.2 Implementation

Due to their advantages, it was decided to use alternative 2 for basic components, such as capacitors and inductors and to use alternative 3 for all other multi-level components (using the new Modelica feature of conditional declarations). Also test equipment, such as load resistances as function of time, may depend on the accuracy level as well as the connectors. In this section all the details are explained including library structuring.

The basic approach is to use multi-level equipment components in a system model and select in a global menu the default accuracy level. At every instance of an equipment model it is possible to define whether the default accuracy level shall be used (= default behavior) or another one via the “level” parameter. Possible options are

- “global option”: Use globally defined level
- “level 1”: Use architectural model level
- “level 2”: Use functional model level
- “level 3”: Use behavioral model level

. “outer” references the “global_options” component in the enclosing environment. Equations and/or declarations of a model are activated and deactivated depending on parameter “actualLevel” or the derived binary flags level1active, level2active or level3active. Since “actualLevel” is a parameter expression (= an expression depending only on literals, constants and parameters), Dymola evaluates conditions of if-clauses that depend on “actualLevel” at compile time and therefore selects the corresponding if-branch also at compile time. This means that any change of “actualLevel” requires re-compilation of the system model.

In order to simplify the multi-level model development, some partial models and classes can be reused. Models can inherit the multi-level components like the “outer” parameter “level” to define the desired modeling level via “extends *partialmodel*” . This is demonstrated hereafter with the multi-level model “issue1”:

```
model issue1 "my model with several levels"
  extends mylib.interfaces.partial_3_levels;
  ...
  equation //for base components:
    if level1active then
      ...;
    else
      ...;
    end if;
  public //for hierarchical multi-level models:
    componentsIssue1.architecturalModel ModelLevel1
  if level1active;
    componentsIssue1.functionalModel ModelLevel2 if
  level2active;
    componentsIssue1.behaviouralModel ModelLevel3 if
  level3active;
  ...
end issue1;
```

which extends partial_3_levels:

```
partial model partial_3_levels
  "Parent class that should be included via extend
  for a multi-level model with 3 levels"
  import Choice = mylib.types.level_choice;
  parameter Choice.temp level="global option" ;
  "Model level to use (global setting,
  architectural/functional/behavioral level)"
  protected
    outer mylib.components.global_options
  global_options;
    parameter String actualLevel = if
  (level == Choice.global_options) then
  global_options.defaultLevel else level;
    parameter Boolean level1active = (actualLevel ==
  Choice.level1);
    parameter Boolean level2active = (actualLevel ==
  Choice.level2);
    parameter Boolean level3active = (actualLevel ==
  Choice.level3);
    parameter mylib.components.global_options
  global_options_temp( defaultLevel=level);
  end partial_3_levels;
```

The possible choices are defined in level_choice:

```
package level_choice
  constant String global_options="global option";
  constant String level1="level 1";
  constant String level2="level 2";
  constant String level3="level 3";
  type temp
    extends String;
    annotation (choices(
      choice="global option" "use global option
  setting",
      choice="level 1" "level 1 (architectural
  level)",
      choice="level 2" "level 2 (functional level)",
      choice="level 3" "level 3 (behavioral
  level)"));
  end temp;
end level_choice;
```

With the future implementation of Modelica enumerations in simulation environments the need for complex

naming of the choices will become obsolete.

The “global_options” object which defines the global setting has to be dragged to the highest hierarchy level and set to "inner". This defines a "global" structure that is accessible by all components on the same or a lower hierarchical level. To generate the “inner” object automatically when dragging (“inner mylib.global_options global_options;”), the following declaration is used (this property is defined via an annotation in the model):

```

model global_options
  "Global options settings"
  annotation (defaultComponentName="global_options",
    defaultComponentPrefixes="inner",
    missingInnerMessage="A \"global_options\"
    component was introduced with default options.",
    ...
  );
  parameter mylib.types.level_choice_default.temp
  defaultLevel= "level 1" "Default model level
  (architectural/functional/behavioral level)";
end global_options;
  
```

Almost identical to level_choice, the global_options object supports choices for the three levels but of course no choice for global_option itself:

```

package level_choice_default
  constant String level1="level 1";
  constant String level2="level 2";
  constant String level3="level 3";
  type temp
    extends String;
    annotation (choices(
      choice="level 1" "level 1 (architectural
      level)",
      choice="level 2" "level 2 (functional level)",
      choice="level 3" "level 3 (behavioral
      level)"));
    end temp;
  end level_choice_default;
  
```

As explained earlier, the multi-level container object for the alternative 3 has to be the superset of the connectors of the single models. To avoid unnecessary variables, the concept of “expandable connectors” can be employed (e.g. expandable connector positive_plug_expandable "Positive expandable electric plug"). The content of this Modelica connector is defined by the sum of connected variables. With this, non identical connectors of the level-dependent models may be connected to the container object connector but only the data of the level selected are present after compilation. E.g. for the transformed/not transformed three phase system, dq transformed components demand one extra variable in the connector: the rotor angle. On the other hand just the dq system uses 2 current/voltage connectors while the non transformed abc system uses three of them.

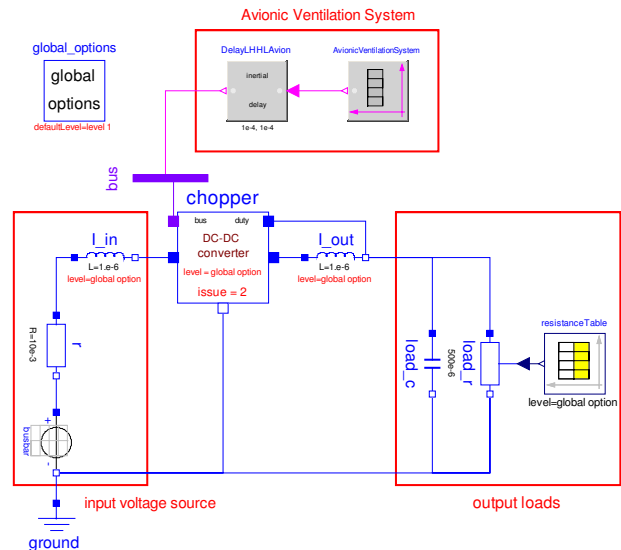


Figure 2: Test circuit for chopper use case (for all 3 levels)

4 Demonstration

In the following chapter the library concept shall be demonstrated with a generic chopper model (electrical non-isolated DC/DC buck converter). The converter is used for transformation of the 270 Volts DC supply to a 28 VDC load network. It was designed to include the most important obstacles, e.g. different levels have different interfaces, components, and load profiles. The behavioral model contains nonlinear switching semiconductors. Only with this the ripple detection for high frequency noise is applicable. The functional model is averaged with the output voltage as the duty ratio times the input voltage. The architectural model lacks of any dynamics. The converter is modeled with an energy balance. A test model for the chopper can be seen in figure 2.

Base components that are solely described by equations are implemented with if-clauses that depend on parameter “actualLevel”. For example, the inductor is defined by:

```

if level1active then v = 0;
else // level 2 or 3
  L*der(i) = v;
end if;
  
```

This means, for level 1 the dynamics equation is removed. In the icon of the inductor, the instance name (here: “inductor”), the inductance (here: “1.2e-6) and the value of parameter “level” (here: global option) are displayed.

Other components are implemented with container objects and conditional declarations as sketched in alter-

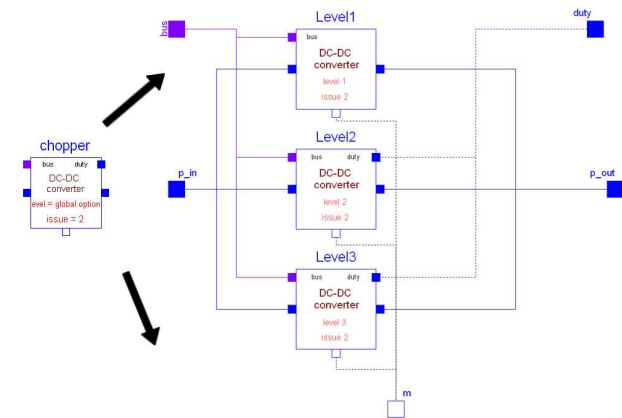


Figure 3: Multi-level chopper component with conditional declaration

native 3 above. So, the structure of the chopper model can be seen from figure 3.

In the left part of the figure, the icon of the chopper is shown. Besides the model instance name, the value of parameter “level” is displayed in the icon. In the right part of the figure, the container object is present. It contains models of every level. All models are connected to the connectors (p_in, p_out, m, duty, bus) defined in the icon and are defined by “conditional declarations”:

```

model issue2
  extends mylib.utilities.interfaces.partial_3_levels
  parameter Modelica.SIunits.Voltage v_out_desired
    = 28;
  ...
  mylib.chopper.componentsIssue1.level1 Level1(
    v_out_desired = v_out_desired) if level1active;
  mylib.chopper.componentsIssue2.level2 Level2
    ( v_out_desired = v_out_desired, k_integrator
    = k_integrator, L_filter = L_filter, R_filter =
    R_filter) if level2active;
  ...
  equation
    connect(Level1.p_out, p_out);
  ...
end issue2;
    
```

For example, component “Level1” is an instance of “mylib.chopper.comonentsIssue1.level1” and is only present if “level1active=true”. If a component is used in a connection, such as “connect(level1.p_out, p_out)”, this connect statement is automatically removed if one or both of the models referenced in the connect(..) statement are deactivated. Therefore, it is no longer necessary to manually include an if-clause around such connect(..) statement as in previous versions of Modelica. Note that the interface of level 1 does not include the “duty” pin but for the multi-level container object the pin is skipped if the level is level

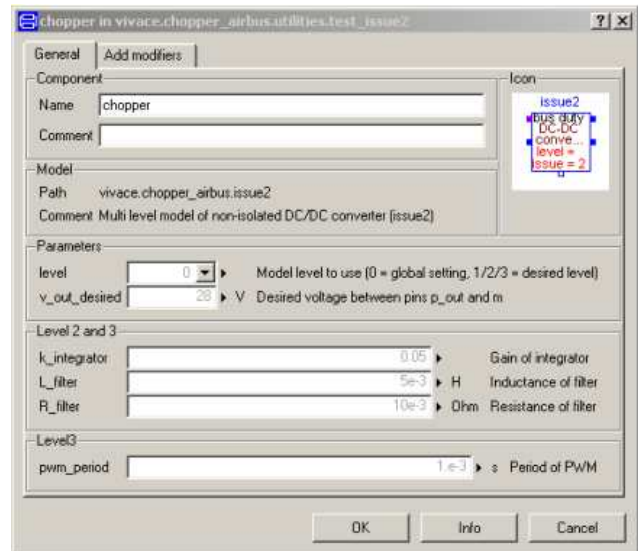


Figure 4: Menu of multi-level chopper component

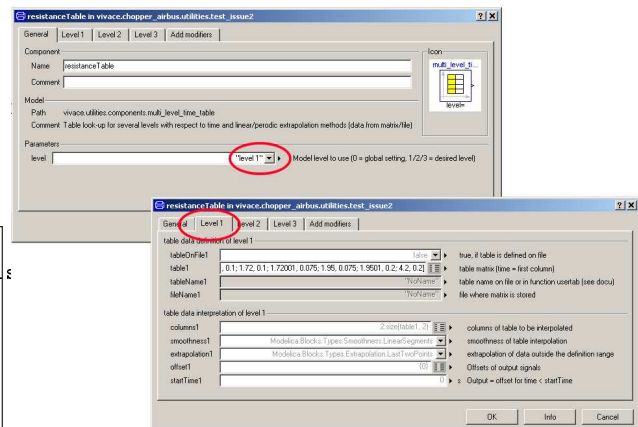


Figure 5: Menu of time_table component

1. It is convenient for the user if the parameters of all levels are defined as parameters of the container object and are propagated to the corresponding models. For example, when clicking on the icon of the chopper example above, the menu shown in figure 4 is opened. When using model level 1, only parameter “v_out_desired” is actually utilized. For model level 3, all parameters in this menu are taken into account. If the parameters of the levels are different, it is useful to display them in different “tabs”. For example, in the menu of component “mylib.utilities.components.multi_level_time_table” shown in figure 5, different tables can be defined for the various levels (the purpose is, e.g., to have different load resistances for the various levels).

5 Application

Without going into detail, this chapter is a short outlook on the usage of the models for stability and quality studies.

A regulated buck converter is a typical critical component in a power system. Due to the negative resistance at low frequencies the regulated buck converter could be unstable in combination with the input filter. Therefore it is necessary to investigate the stability of the whole electric network both at small signal level for steady state conditions and large signal level for transients, impacts and network reconfiguration. The library concept proposed above is a good tool for its usage since validity of functional models can first be demonstrated by comparison of simulation results with the behavioral models. The Modelica functional models can be used for stability studies with methods for linear time invariant systems. For eigenvalue based methods, including modal analysis and eigenvalue sensitivity, the eigenvalues have to be calculated numerically by the simulation program. μ analysis is a powerful method of robust control for stability investigations of parametric varying systems. The Modelica functional models can be directly applied for μ analysis after extraction of the symbolic code and using it in Maple and Matlab. Compared with other methods for small signal stability, e.g. Middlebrook criterion and Modal Analysis, the μ sensitivity approach gives a much more global and direct result for the influence of all components on stability. For details on the methods, see [2].

In contrast to these approaches, for industrial use stability of a system often is defined as the ability of a system to keep a certain system variable within desired limits given by industrial standards. These are combined criteria of network stability, power quality and performance. This makes them difficult to proof with methods of linear control theory. Therefore a simulation based approach often is the only possibility to proof "industrial" stability and also large signal stability including failure protection devices. Instead of random or gridded parameter variation on the varying environment and system parameters, an other method is to search for the most critical parameter combination directly. The basic idea is to use an optimizer to find the criterion from the standards which is most critical and make it worst by changing the uncertain parameters in the possible range. In case the criterion is violated, stability/quality/performance can be shown to be not guaranteed. On the other hand, the tolerable design range for parameters could be investigated as the

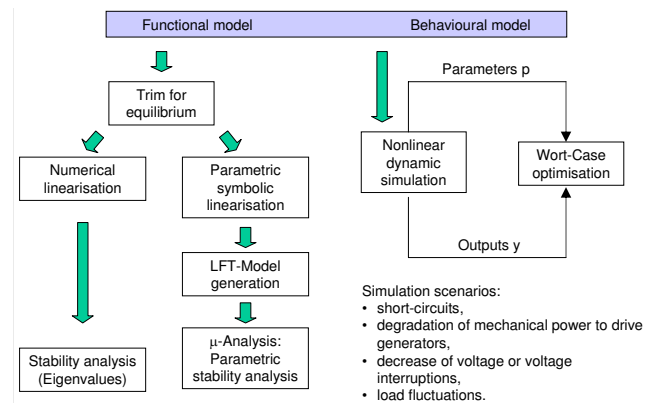


Figure 6: Quality and stability studies overview

bounds leading to standard violations.

An overview on the methods is shown in figure 6.

6 Acknowledgment

This work was in parts supported by the EU in the "VI-VACE project" (<http://www.vivaceproject.com/>) under contract number 502917.

7 Conclusion

In this paper a multi-level concept used for aircraft electrical systems design based on conditional declarations was shown. The three level concept was explained and modeling demands and methods were proposed, especially using park transformation for AC systems. The implementation was demonstrated with a multi level DC/DC chopper use case. The modeling concept improves modeling of large systems and allows easy comparison of different levels simulation result. An overview on typical applications of the models for stability and quality studies was given..

References

- [1] Paul C. Krause, Oleg Wasynczuk, and Scott D. Sudhoff. *Analysis of electric machinery and drive systems*. Wiley-Interscience, Piscataway, New York, 2nd edition, 1998.
- [2] M.R. Kuhn, Y. Ji, and D. Schröder. Stability studies of critical dc power system component for more electric aircraft using mu sensitivity. In *Proceedings of the 15th Mediterranean Conference on Control and Automation*, Athens, 2007.

Incorporation of Reliability Analysis Methods with Modelica

Christian Schallert

German Aerospace Centre (DLR), Institute of Robotics and Mechatronics
82234 Wessling, Germany
Christian.Schallert@dlr.de

Abstract

A novel method is being developed to combine techniques of safety and reliability analysis with the Modelica language, which is now widely used for the modelling and simulation of technical systems.

The new method allows to perform reliability calculations based on the system model that is created and used for simulation studies. The reliability analysis procedure is started “at the push of a button” and determines the so called minimum path sets and the failure probability of a system automatically.

The incorporated reliability computation methods are realised initially by a new modelling and analysis tool supporting concept design studies of aircraft on-board electric power systems.

Keywords: reliability, fault modelling; model object structure; minimum path set; failure probability

1 Introduction

Much of the information needed for reliability calculations is contained already in compound system models that are usually built in Modelica [1]. The specific modelling additions needed, as well as the fundamentals of an automated reliability analysis procedure are described by this paper.

The procedure evaluates the physical behaviour of a modelled system in multiple simulations. An addition needed to the modelling is the faulty behaviour of components, as described in chapter 2.1.

Prior to evaluating the system model by numerous simulations, its object structure is appraised in order to detect those combinations of components, that represent candidates of so called minimum path sets. Chapter 2.4 gives an overview of the method and definitions, as well as a way of minimising the computing effort that is involved with this kind of automated reliability analysis.

2 Modelling Approach and Integrated Reliability Analysis Concept

2.1 Component Fault Modelling

A variety of object-oriented model libraries have been developed in the Modelica language, as generally known. In each component model, the normal operational behaviour is described by differential and/or algebraic physical equations.

For the purpose of performing reliability analyses, the component models have to be enhanced such that also the failure behaviour is described by physical equations. Basic examples are given hereafter by the modelling assumptions made for some common electrical components:

An electric wire can be described as an ohmic resistor. For the normal function of the wire, its nominal resistance R_{nom} is in the order of $10^{-1} \Omega$. An open circuit failure of the wire is characterised by a very large resistance, e.g. $10^6 \Omega$.



Figure 1: Modelica Object Diagram of an Electric Resistor

In essence, the following code defines the model:

```
model Resistor "Ideal linear resistor"
  Interfaces.Electrical.PositivePin p;
  Interfaces.Electrical.NegativePin n;
  input Boolean FAILED;
  parameter Real lambda = 2e-5 "failure
    rate";
  parameter SI.Resistance Rnom = 0.1;
  SI.Resistance R = if FAILED then 1e6
    else Rnom;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  R*i = v;
end Resistor;
```

A generator can be represented by its DC substitute properties and the efficiency of converting mechanical into electric power. A basic description of a generator failure is the loss of output voltage, which stems from an internal failure of the generator or from insufficient generator drive speed.



Figure 2: Modelica Object Diagram of a Generator

The following code basically defines this model:

```
model Generator "DC generator with losses"
  Interfaces.Rotational.Flange_a
    flange;
  Interfaces.Electrical.DC_Plug_a plug;
  input Boolean FAILED;
  parameter Real lambda = 1e-4 "failure rate";
  parameter SI.Power Pnom = 5e4;
  parameter SI.Voltage Vnom = 270;
  NonSI.AngularVelocity_rpm speed;
  SI.Voltage v;
  SI.Power Pelec;
  SI.Power Plosses;
  SI.Power Pmech;
equation
  speed = max(0.1 ,
    to_rpm(der(flange.phi)));
  v = if FAILED then 0 else (1-
    exp(-speed/1000))^2*Vnom;
  v = plug.pin_p.v - plug.pin_n.v;
  Pelec = v*plug.pin_n.i;
  Plosses = 3000*speed*Pelec/(14600*
    Pnom);
  Pmech = Pelec + Plosses;
  Pmech = -flange.tau*from_rpm(speed);
end Resistor;
```

Each component model has a boolean input signal *FAILED* to control its status, i.e. operation or failure. The status can be shifted during simulation. Failure rates *lambda* are stored in each component model as modifiable parameters. Using constant failure rates is adequate w.r.t the assumption of an exponentially distributed component lifetime. Other hypotheses on the dependency of failure rates on lifetime can also be taken into account.

Thus, a new Modelica library of electric component models, that are augmented with a basic failure behaviour, is being developed. In doing so, the fundamental concept of creating component models that are usable regardless of the application case or physical context, is being followed. Compatibility with existing model libraries is maintained as well.

2.2 Integrated Tool Concept

The new library of electric component models, as well as integrated reliability analysis procedures are part of a new developed concept design tool for aircraft on-board electric power systems. Besides reliability, the tool is prepared to evaluate architecture concepts w.r.t. the electric behaviour and weight, as illustrated by Figure 3.

Large compound models of electric power systems can be assembled using the graphical model editor of Modelica/Dymola [2] in the known fashion.

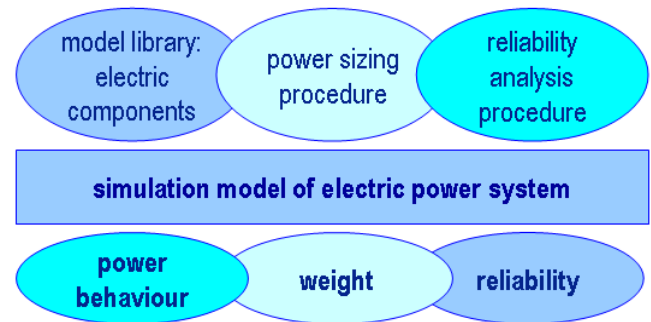


Figure 3: Elements of Modelica based Concept Design Tool for Electric Power Systems

Aircraft electric power systems are of particular interest for reliability analysis, since they supply a multitude of loads, many of which fulfil a function that is essential for safe flight and landing. Also, the electric demands tend to increase, due to the recent trend in the design of transport aircraft to replace hydraulic and pneumatic supplies by electric power [3].

Electric power systems on aircraft are typically split into several independent channels, each comprising an engine driven generator, a distribution network and a number of loads. If failures occur, the electric power system is reconfigured automatically to isolate the fault and to secure power supply to most of the loads, with priority to the essential ones. The redundancies and reconfiguration capability of such systems have to be included in the system model accordingly, by means of open/close logics for the electric network contactors. Thus, the behaviour in various operational scenarios, e.g. normal, abnormal or emergency, can be examined in simulations.

2.3 Modelling Example: Electric Power System

In the following, the integrated modelling and reliability analysis concept is illustrated by the example of an electric power system of a generic twinjet aeroplane. The system model, see Figure 4, has been devised based on a description available in [4].

2.3.1 System Model Features

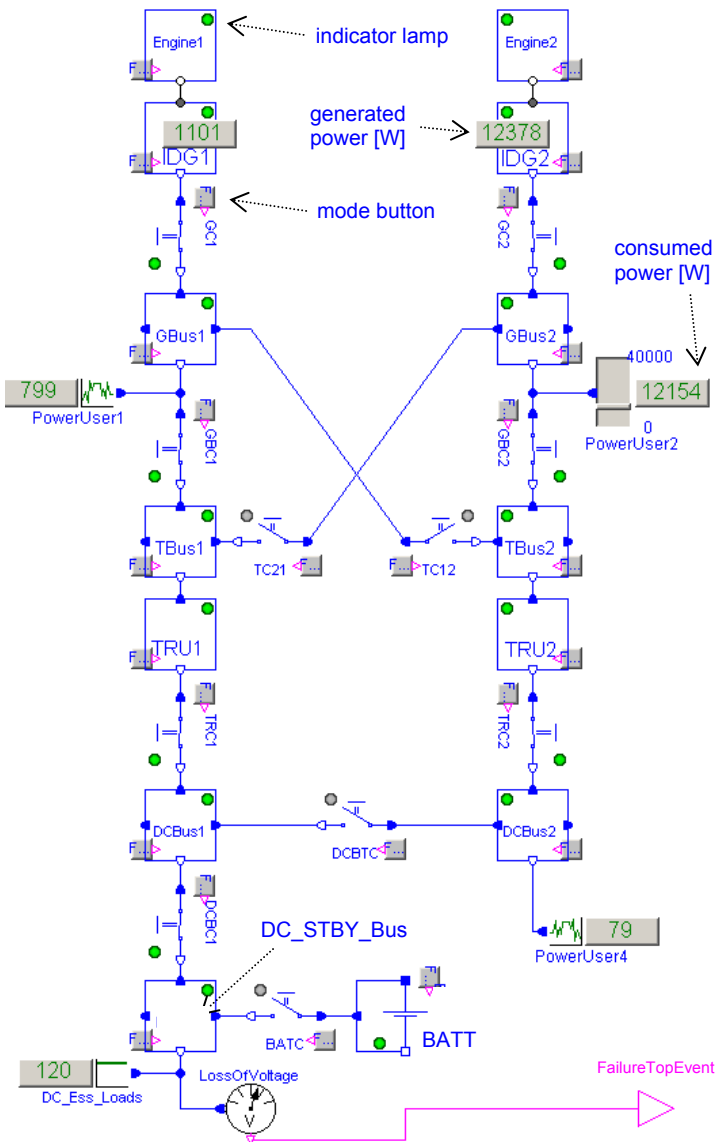


Figure 4: Electric System Model of a Twinjet Short-Range Aeroplane, schematic shows Normal Operation in flight

Figure 4 also shows the object-oriented structure of the system model, which is commensurate with the general philosophy of Modelica. Note that the electric connections include two poles.

The electric system model includes the following, most important components and features:

- Two integrated drive generators *IDG1* and *IDG2*, which are driven by *Engine1* and *Engine2*, respectively. Each generator provides 115V / 400Hz AC power (substituted in the model by 270V DC)
- through a dedicated generator contactor, *GC1* and *GC2*
- on the generator buses, *GBus1* and *GBus2*.
- The main non-essential AC loads, represented by *PowerUser1* and *PowerUser2*, as well as
- the AC transfer buses, *TBus1* and *TBus2* are connected to the respective generator buses. In normal operation with both engines running (as shown in Figure 4), each transfer bus is supplied by its associated generator bus through a generator bus contactor, *GBC1* and *GBC2*.
- The AC cross transfer contactors, *TC12* and *TC21*, are open in this normal operating case.
- 28V DC power is provided by two transformer rectifier units, *TRU1* and *TRU2*, through dedicated switches *TRC1* and *TRC2*, on the DC busbars *DCBus1* and *DCBus2*.
- The two DC busbars can be cross-connected through the *DCBTC* switch. In normal operation (Figure 4), the cross-connection is inactive i.e. the *DCBTC* switch is open.
- Finally, a stand-by busbar *DC_STBY_Bus* provides for the essential loads *DC_Ess_Loads*, which must operate even after a complete loss of generated power, to maintain safe flight and landing. In such a scenario, the essential loads are powered by a battery *BATT* through the *DC_STBY_Bus*.

The following apparent features are included in the system model:

- Each contactor has an animated rocker switch to depict its open / closed status.
- Each component model is fitted with an indicator lamp. During simulation of the model, the operational (green), passive (grey) or failed (red) status of each component is shown by the associated indicator lamp.

A component is defined as operational when turning (e.g. engine), energised with voltage (e.g. busbar) or conducting current (e.g. switch), whatever is applicable. The passive status is specified as the component being intact but not energised. If a component

has failed, it cannot be energised with voltage or conduct current, whatever is applicable.

Each component model is provided also with a mode button for an interactive control of its operative / failed status. Pressing the mode button of a component, by mouse-click, during simulation will toggle the status (operational or passive \leftrightarrow failed) of the component.

By means of the mode buttons and indicator lamps that are provided with the component models, the behaviour of a system model can be examined interactively during simulation. This is useful when developing the network switching logics, since the resulting behaviour at system level can be checked quickly and readily.

Furthermore, the following features are included in the system model:

- Generic power users, which can be connected to any busbar as needed. The power users are described by basic resistive properties, which can be set by parameter entries or interactively: The *PowerUser2* shown in Figure 4 has an adjustable slider bar, which is set by mouse-dragging during simulation.
- Each power user model, as well as the generator (IDG) models have a numerical indication of the generated / consumed power in W(att).
- A *FailureTopEvent* definition at system level [5], so that the system reliability w.r.t. this event can be computed. In the present example shown by Figure 4, the system failure top event is defined as a loss of voltage on the *DC_STBY_Bus*.

As can be seen in Figure 4, the definition of the system *FailureTopEvent* has been implemented by connecting a specific voltage sensor named *LossOfVoltage* to the *DC_STBY_Bus*. The sensor will flag a voltage drop-out below a defined threshold by its logical output signal. This specific voltage sensor model class is provided by the model library.

Other system failure event definitions are conceivable, e.g. a loss of voltage on other busbars or combined events, such as the loss of voltage on *DCBus1* and *DCBus2*. Any meaningful failure event definition can be implemented in an accordant manner, by use of the provided sensor class and logical gates.

2.3.2 Degraded System Operation

In Figure 5, a degraded operational mode of the electric system, caused by failures of *Engine2* and *TRU1*, is shown. The component failures have been injected

in the model by pressing the corresponding mode buttons during simulation.

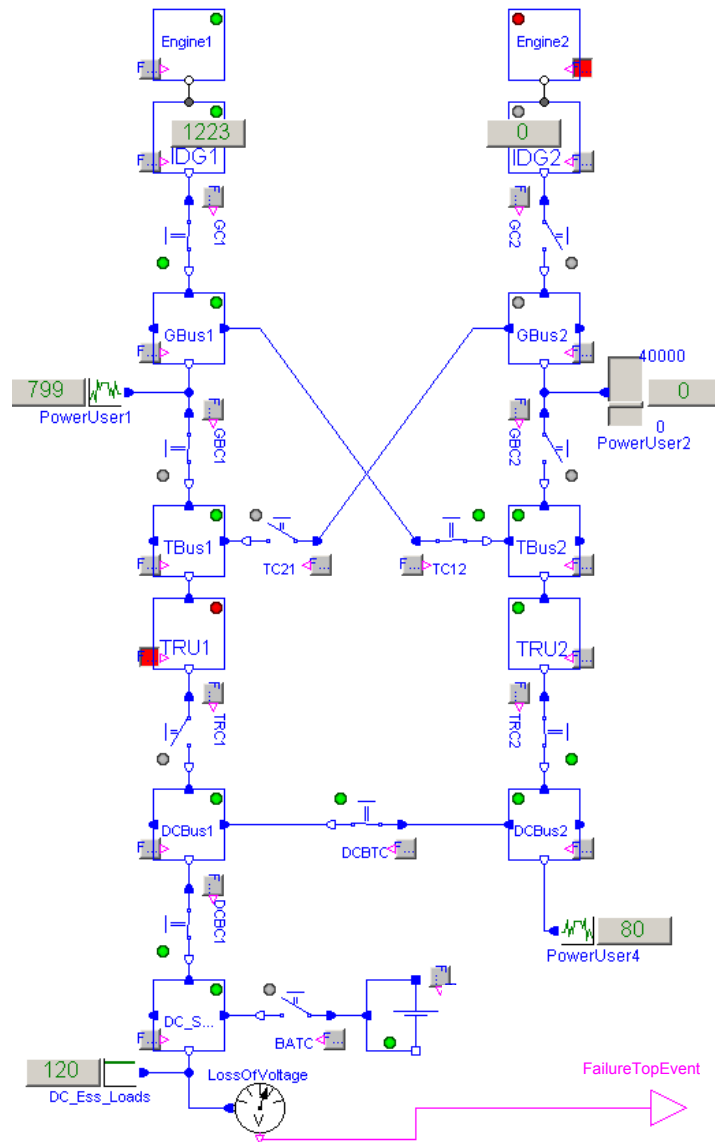


Figure 5: Twinjet Aeroplane Electric System Model, Engine2 and TRU1 have failed

The failure effects are, as can be seen in Figure 5:

- *IDG2* does not operate. Consequently, the *GBus2* is de-energised, as well as the connected *PowerUser2*.
- *TBus2* is now energised by the opposite side through the *TC12* switch, which has been closed automatically.
- All DC busbars *DCBus1*, *DCBus2* and *DC_STBY_Bus* are now supplied by *IDG1* through *TRU2*.
- Although degraded, the system is still operational in the complementary sense of the defined *FailureTopEvent*.

2.4 Reliability Analysis Procedure

The reliability analysis procedure is capable of automatically determining the so called minimum path sets for a given system model. A minimum path set is a combination of operative components that causes a system to operate in the complementary sense of the defined failure top event. Further on, the procedure computes reliability measures, i.e. system failure probability w.r.t. the defined top event, as well as component importances. For further illustration, results are shown in chapter 2.5 for the modelling example introduced in chapter 2.3.

The reliability analysis procedure draws on two kinds of information contained in a system model, as will be depicted: Chapter 2.4.1 explains a method to evaluate the system model behaviour in terms of operation or failure by multiple simulations. Then, chapter 2.4.2 introduces a method to interpret the object structure of the system model. Finally, chapter 2.4.3 describes how the two methods are combined to an automated reliability analysis procedure.

2.4.1 System Model Evaluation by Simulations

A simulation based method evaluates the system model for combinations of operative and failed components in a specific order. Minimum path sets are detected by the occurrence of system operation, i.e. the logical signal *FailureTopEvent* flagged as false.

Each system model contains the n components C_1, C_2, \dots, C_n . At first, the system model is simulated for single ($k = 1$) intact components. Each row 1, 2, ..., n in Table 1 represents one set of operative (OK) and failed (-) components to test in the simulation.

Table 1: Intact/Failed Components to Simulate, $k = 1$

	C_1	C_2	...	C_n
1	OK	-		-
2	-	OK		-
...			...	
n	-	-		OK

If the system is operational for a row of Table 1, then the intact component of that row is stored as a minimum path set.

The method continues with simulating for two ($k = 2$) intact components, as depicted in Table 2. Again, each row stands for one set of operative and failed components to test. If minimum path sets of lower order ($k < 2$) were found, then those rows of Table 2 that contain all intact elements of a previously de-

tected minimum path set are not tested in the simulation. This way, it is ensured that each detected path set is minimum, meaning that it does not contain any subset of other path sets.

Table 2: Intact/Failed Components to Simulate, $k = 2$

	C_1	C_2	...	C_n
1 2	OK	OK		-
...			...	
1 n	OK	-		OK
...			...	
2 n	-	OK		OK
...			...	

If the system is operational for a row of Table 2, then the intact components of that row are stored as a minimum path set.

In an analogous manner, the method continues with the determination of minimum path sets by simulating the system model for intact components up to an order of $k = n$, see Table 3.

Table 3: Intact/Failed Components to Simulate, $k = n$

	C_1	C_2	...	C_n
1 2 ... n	OK	OK	...	OK

Apparently, this simulation based method has a character of systematic trial and error. Yet, the computing effort increases significantly with the number of components contained in a system model. For a system model comprising n components, a total of up to N sets (rows) have to be checked by simulations:

$$N \leq \sum_{k=1}^n \binom{n}{k}$$

Table 4: Estimation of Computing Effort

n	1	2	3	4	...	10	...	20
N	1	3	7	15		1023		1048575

Consequently, this method of minimum path set determination is only practical for systems including relatively few components. On its own, this method is not suitable for analysing the example model shown in Figure 4, which represents an electric system including 25 components.

So far, the system model is checked only in simulations. A further possibility is to evaluate the object structure of the system model, as described in 2.4.2.

2.4.2 Object Structure of the System Model

Another method exploits the object structure of the system model, i.e. the arrangement of components and connections. Advantage is taken of the fact that the structure of object-oriented models is similar, although not exactly identical, to minimum path sets.

Thus, a specific algorithm is devised to analyse the succession of connected components. As a result, the algorithm yields the different paths of consecutive and non-repeating components that exist in a system model. The paths that are determined in this manner are considered as minimum path set candidates.

The fundamentals of this kind of algorithm are described hereafter. It is realised as a recursive model parser in Modelica. In the listing, the notations component1, component2 and path indicate variables.

1. Begin at the FailureTopEvent gate of the system model and add it as component1 to the path.
2. Find all components connected to component1.
3. If no components are connected to component1 then terminate the actual recursion branch.
4. If one component is connected to component1 then take it as component2 and continue with the actual recursion branch,
5. else if more than one components are connected to component1 then start a new recursion branch for each component taken as component2, respectively.
6. If component2 is not contained in path yet then add component2 to path and resume at step 2 taking component2 as the next component1,
7. else terminate the actual recursion branch.

The result of this system model object structure analysis are paths that are considered as minimum path set candidates. These are illustrated graphically in Figure 6 for the electric system introduced in chapter 2.3. A representative selection of the 29 paths determined for this example is shown.

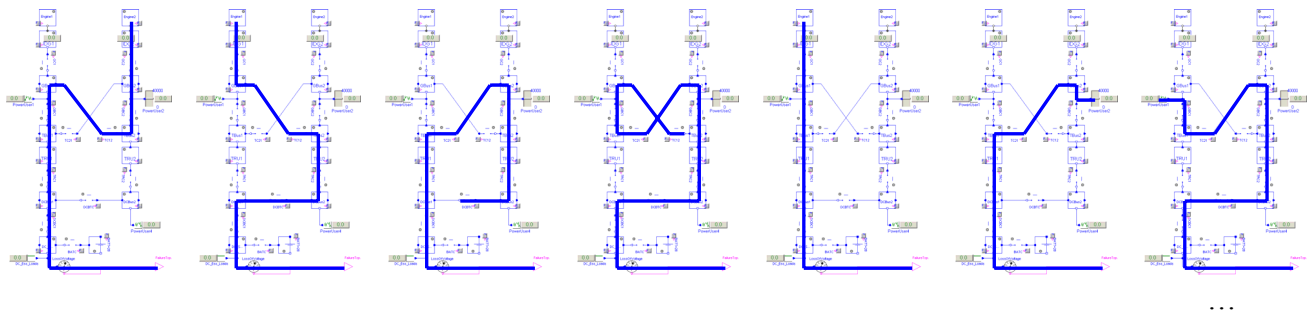


Figure 6: Model Object Structure Analysis: Graphical Representation of Several Minimum Path Set Candidates

2.4.3 Combination of Object Structure Analysis with Simulation Based Method

As mentioned, the found paths are considered as minimum path set candidates. Therefore, these candidates are checked by simulating the system model accordingly, to eventually extract the minimum path sets from the list of candidates.

In this method, the system model is simulated for each candidate, such that the components belonging to a candidate are switched to the intact mode one after another, while all other components of the system are failed. System operation or failure is detected in the simulation by evaluating the logical signal *FailureTopEvent*. If the system operates, then the causing set of intact components is stored as a minimum path set.

The number of path candidates to be checked in the simulation is limited, hence conducting an object structure analysis first and then simulation minimises the overall computing effort. Thus, the combination of both leads to a reliability analysis procedure that is viable even for large systems with many components.

After the minimum path sets of a system have been determined, reliability measures can be computed.

The probability of occurrence belonging to each minimum path set, i.e. the system operates, MP_i is

$$P(MP_i) = \prod_{C_i \in MP_i} (1 - p_i), \text{ with the components } C_i \text{ and}$$

the individual failure probabilities p_i .

Assuming an exponentially distributed lifetime [6] for the components C_i leads to failure rates λ_i that are constant over lifetime. The probability of a component failure is

$$p_i(t) = \begin{cases} 1 - e^{-\lambda t}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

Since the occurrence of at least one minimum path set causes the system to operate, the probability of system operation can be calculated according to Poincaré's formula [6] as

$$\begin{aligned}
 P_{\text{system-operation}}(p_i) &= P(MP_1 \vee MP_2 \vee \dots \vee MP_n) \\
 &= \sum_{j=1}^n (MP_j) - \sum_{i=1}^{n-1} \sum_{j=i+1}^n P(MP_i \wedge MP_j) + \dots \\
 &\quad + (-1)^{n+1} P(MP_1 \wedge MP_2 \wedge \dots \wedge MP_n)
 \end{aligned}$$

with n being the number of minimum path sets.

Generally, i.e. for a single component or a complex system, the following equation holds

$$p_{\text{failure}}(t) + p_{\text{operation}}(t) = 1$$

which eventually allows to calculate the probability of system failure.

Another useful reliability measure are component importances, which help to identify potential weak points or unnecessary redundancies in a system. Several definitions of importances exist. Here, the definition of marginal importances, that indicate the structural and probabilistic influence of a component i in a system, is given by

$$I_{\text{marg}}(i) = \frac{\partial P_{\text{system-operation}}(p_i)}{\partial p_i} \quad \text{with } 0 \leq I_{\text{marg}}(i) \leq 1$$

To summarise, Figure 7 gives an overview of the entire concept of incorporating a reliability analysis procedure with the Modelica language.

2.5 Reliability Analysis Results for Modelling Example

To illustrate the reliability analysis procedure, results are shown below for the modelling example of 2.3.

Table 5 lists the components that appear in the system model with the related failure rates λ_i .

The exposure time is set to $t = 1\text{h}$ for simplicity, so that the failure probabilities are: $p_i(t) \approx |\lambda_i|$

Table 5: Electric System Model Components List

i	C _i	λ _i [1/h]
1	Engine1	10 ⁻⁵
2	Engine2	10 ⁻⁵
3	IDG1	10 ⁻⁴
4	IDG2	10 ⁻⁴
5	GC1	10 ⁻⁶
6	GC2	10 ⁻⁶
7	GBus1	10 ⁻⁷
8	GBus2	10 ⁻⁷
9	TBus1	10 ⁻⁷
10	TBus2	10 ⁻⁷
11	GBC1	10 ⁻⁶
12	GBC2	10 ⁻⁶
13	TC12	10 ⁻⁶

i	C _i	λ _i [1/h]
14	TC21	10 ⁻⁶
15	TRU1	2·10 ⁻⁴
16	TRU2	2·10 ⁻⁴
17	DCBus1	10 ⁻⁷
18	DCBus2	10 ⁻⁷
19	TRC1	10 ⁻⁶
20	TRC2	10 ⁻⁶
21	DCBTC	10 ⁻⁶
22	DC_STBY_Bus	10 ⁻⁷
23	BATT	0.001
24	BATC	10 ⁻⁶
25	DCBC1	10 ⁻⁶

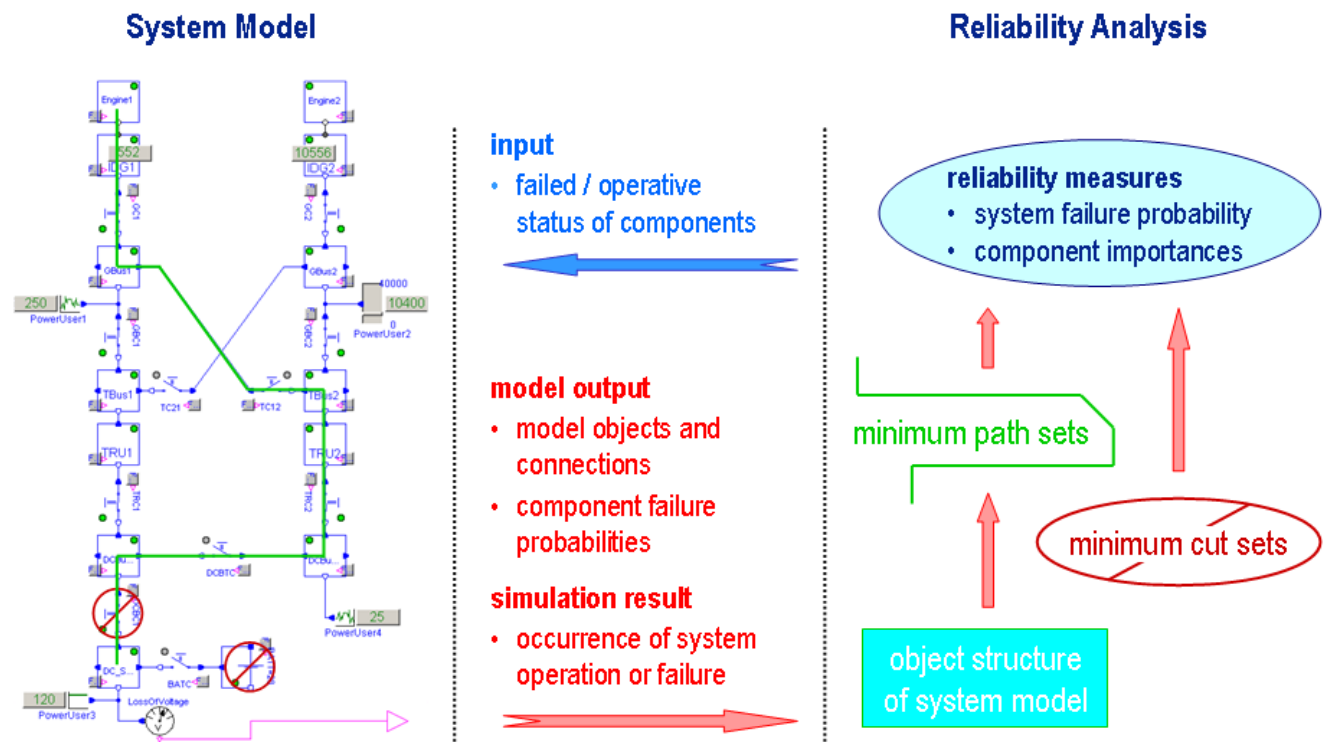


Figure 7: Concept of Reliability Analysis Incorporation with Modelica

2.5.1 Scenario 1: DC STBY Bus Energised

In Figure 8, the five minimum path sets determined for the electric system and the scenario “DC_STBY_Bus energised”, which is the complement of “loss of voltage on DC_STBY_Bus”, are graphically shown. The system operates, i.e. the DC_STBY_Bus is energised, if all components of either minimum path set are operational. For the opposite case, the probability of system failure is computed as

$$P_{system-failure} = 1.012 \cdot 10^{-7}$$

Figure 9 shows a plot of the component importances. Altogether, the analysis result can be interpreted such that for the given scenario, the system failure probability is dominated by a failure of the DC_STBY_Bus itself, followed by failures of the DCBus1 and the contactor DCBC1. The influences of the three redundant voltage sources IDG1, IDG2 and BATT, that can energise this busbar, are much smaller.

2.5.2 Scenario 2: DCBus1 and DCBus2 Energised

In a different scenario, the loss of voltage on DCBus1 OR DCBus2 (or both) is examined for the same electric system. Figure 10 shows the implementation of this scenario in the system model. In the complementary sense, system operation means that both busbars, DCBus1 and DCBus2, are energised.

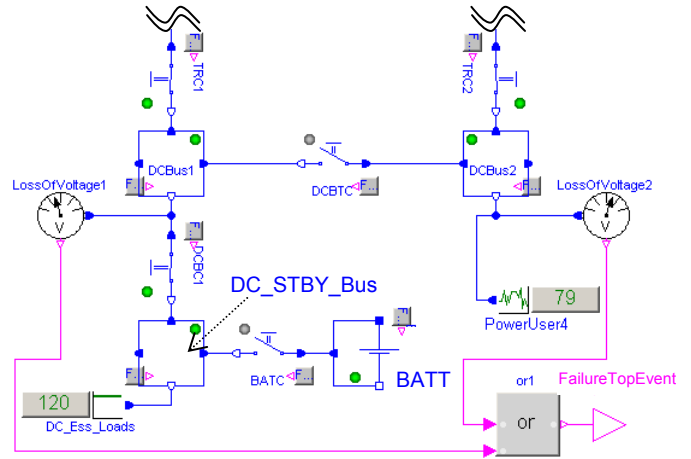


Figure 10: Implementation of Scenario 2 in System Model

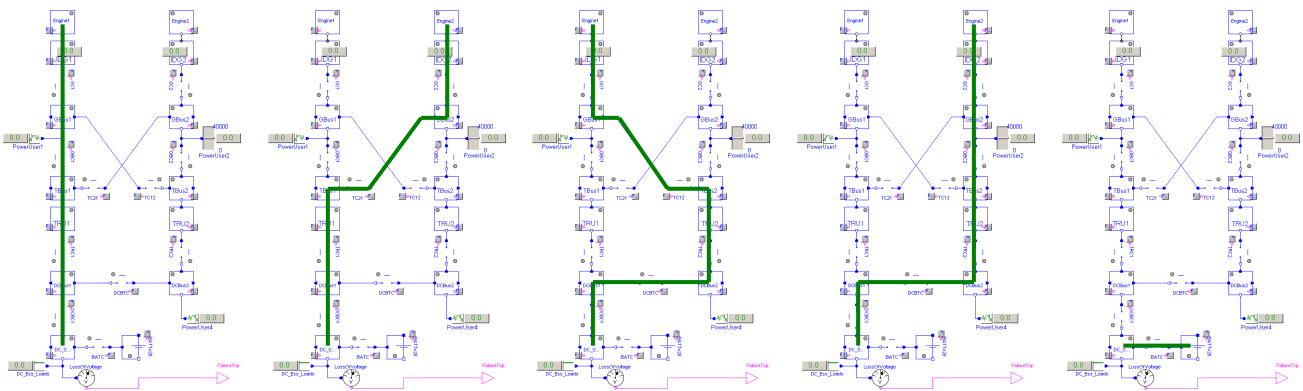


Figure 8: Graphical Representation of the five Minimum Path Sets for Scenario 1 “DC STBY Bus energised”

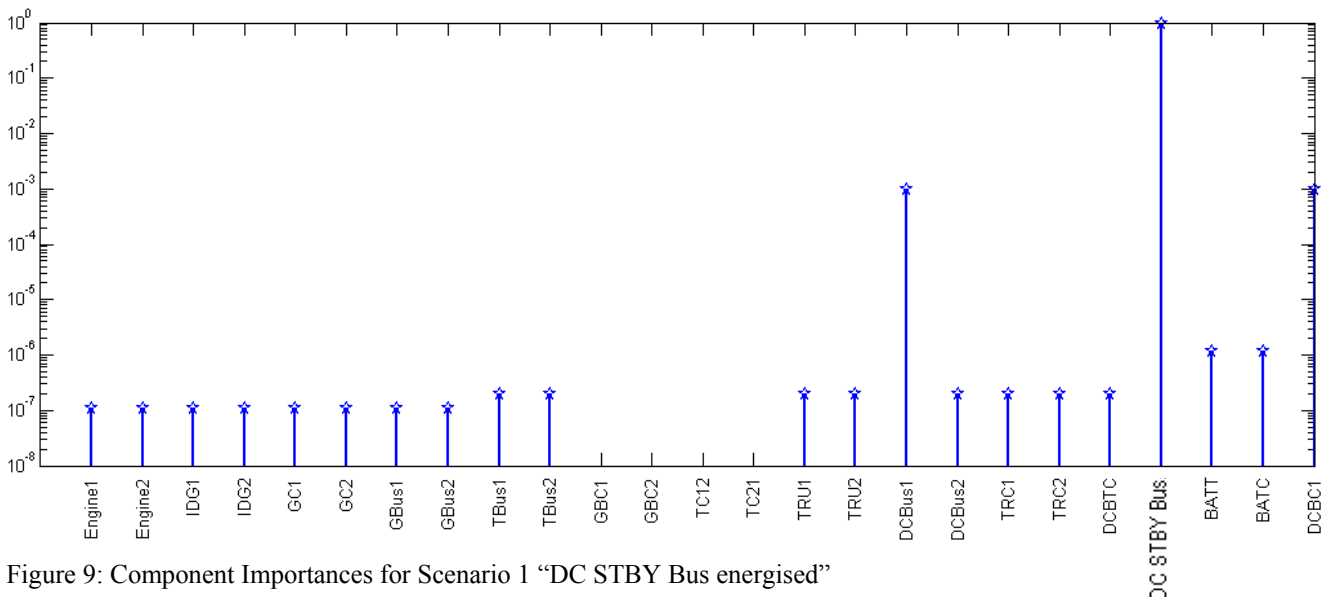


Figure 9: Component Importances for Scenario 1 “DC STBY Bus energised”

In Figure 12, the eight minimum path sets are depicted that were identified for this scenario. The probability of system failure has been calculated as

$$P_{system-failure} = 2.532 \cdot 10^{-7}$$

An interpretation of this result is that again the system failure probability is influenced mostly by busbar failures themselves, as it is indicated also by the importances shown in Figure 11. Other contributions arise from the busbar cross contactor *DCBTC* and from most of the upstream electric network components. The *DC_STBY_Bus* and the *BATT* have no influence since *DCBus1* and *DCBus2* can be energised

only by the generators *IDG1* or *IDG2* through the network.

In summary, the example results demonstrate the capability of the novel reliability analysis procedure incorporated in Modelica to evaluate complex system architectures. The procedure is started “at the push of a button” and automatically computes the results without any further action required from the user.

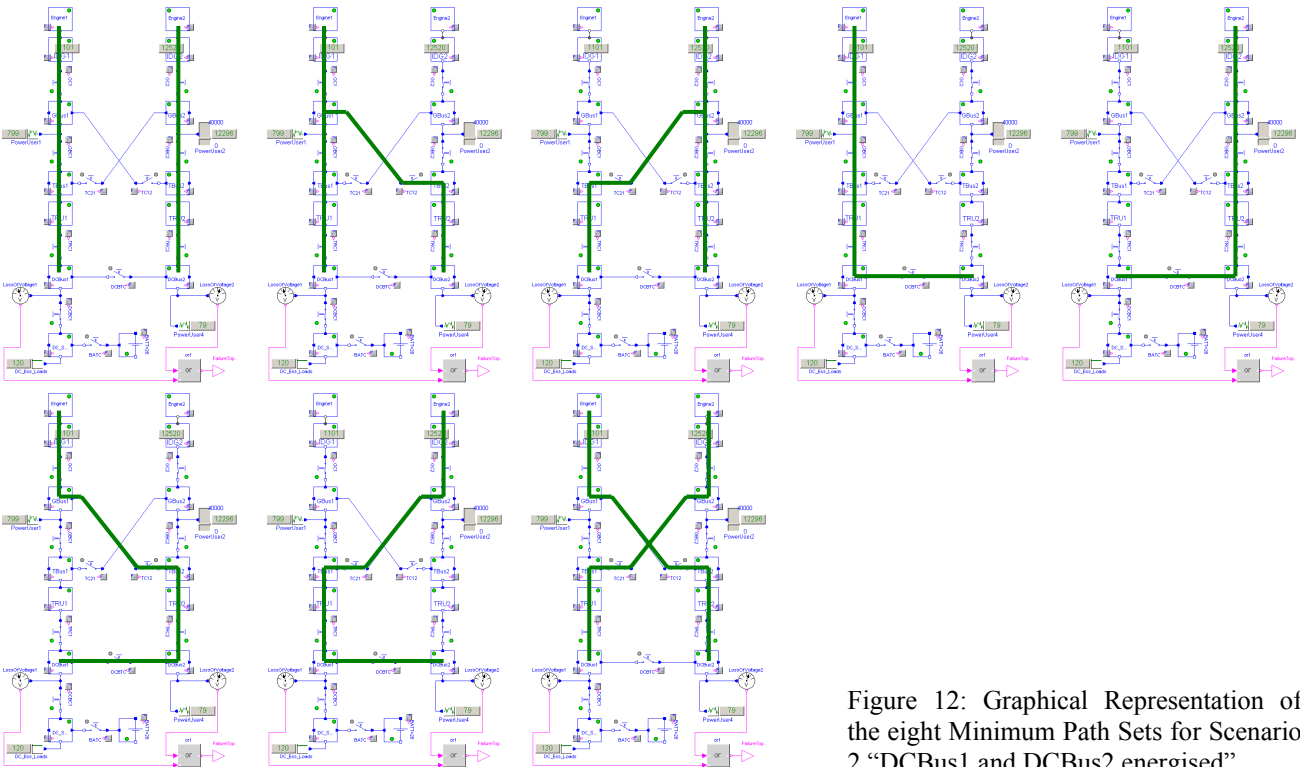


Figure 12: Graphical Representation of the eight Minimum Path Sets for Scenario 2 “DCBus1 and DCBus2 energised”

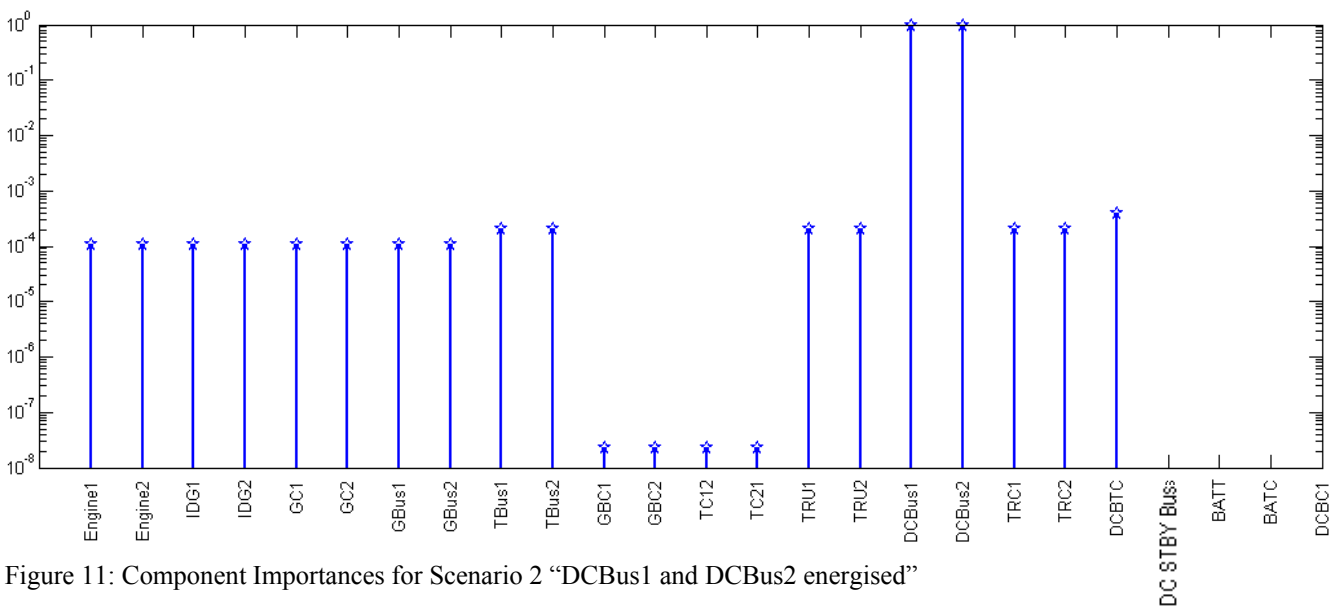


Figure 11: Component Importances for Scenario 2 “DCBus1 and DCBus2 energised”

3 Conclusions and Outlook

A new method to enhance Modelica with the capability of conducting reliability analyses is outlined.

The incorporation of automated reliability analysis methods with Modelica broadens the scope of the language, thus being able to support design studies of redundant and safety critical systems, where sufficient system reliability has to be demonstrated.

The methods are realised initially by a new, Modelica based modelling and analysis tool for aircraft on-board electric power systems. System models can be built and simulated in the known fashion using components from existing and a specific new model library. Then, a reliability analysis can be performed for the same system model “at the push of a button”.

The analysis procedure automatically detects the so called minimum path sets of a system. Further on, reliability measures are computed, like system failure probability, e.g. for the partial or total loss of voltage, as well as component importances. These give insight to potential weakness or unnecessary redundancy that may exist in the design of a system.

Future work will oriented to

- an extension of the reliability analysis procedure, such that it can examine models containing differential equations. The procedure is devised initially for system architecture studies, which are usually carried out on models that solely consist of algebraic equations.
- the creation of an automated power sizing analysis: Minimum path sets represent the different operational scenarios of an electric system, so these scenarios can be evaluated in simulations to determine the maximum power that each component carries [7]. This is affecting the sizing and hence the weight of components. Another possibility is to conduct a power availability analysis, i.e. to compute probabilities for the amount of electric power available on a busbar.
- a widening of the fault modelling, such that each electric component model can be simulated for several kinds of malfunction, e.g. open circuit, short circuit, short circuit to ground etc. This will permit to run so called minimum cut sets analyses, leading to an even more comprehensive assessment of system safety and reliability.
- developing features for an improved graphical representation of the analysis results.
- a transfer of the methods to other physical domains, as well as non-aerospace applications.

Acknowledgements

This research is being conducted in the frame of the MOET project (More-Open Electrical Technologies), a FP6 European Integrated Project [8].

The author wishes to thank the electrical engineering department of Airbus-France for their support being the first beta tester of the tool prototype.

References

- [1] Modelica Language. <http://www.modelica.org>
- [2] Dynasim Dymola. <http://www.dynasim.se>
- [3] Moir, I. and Seabridge A. (2001). *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration*. AIAA Education Series, ISBN 1-56347-506-5.
- [4] Wild, T. W. (1996). *Transport Category Aircraft Systems*. Jeppesen Sanderson, Inc. ISBN 0-88487-232-7.
- [5] Lloyd E. and Tye W. (1982). *Systematic Safety – Safety Assessment of Aircraft Systems*. Civil Aviation Authority (CAA), London, ISBN 0 86039 141 8.
- [6] Meyna A. and Pauli B. (2003). *Taschenbuch der Zuverlässigkeits- und Sicherheitstechnik*. Carl Hanser Verlag München Wien, ISBN 3-446-21594-8.
- [7] Schallert, C. (2007). *A Novel Tool for the Conceptual Design of Aircraft On-Board Power Systems*. SAE AeroTech Congress and Exhibition, Los Angeles, CA.
- [8] MOET Project. <http://www.moetproject.eu>

Simulation of Distributed Automation Systems in Modelica

Florian Wagner Liu Liu Georg Frey
 University of Kaiserslautern
 Erwin-Schrödinger-Str.12
 D-67663, Kaiserslautern, Germany
 {wagner | liuliu | frey}@eit.uni-kl.de

Abstract

The increasing application of network technologies and smart embedded devices in the field of automation and control leads to new distributed system architectures. The analysis of the resulting distributed automation systems requires models that cover physical processes as well as computing and communication devices. Modelica as multi-domain modeling language offers the necessary support to build such models. While, for physical systems, there are many Modelica libraries, only limited support for the modeling of computation and communication is currently available. This gap is filled by the presented network and controller libraries. The network library currently supports switched Ethernet, WLAN, and ZigBee. The controller library offers different types of controllers as well as interface devices. Implementation aspects of the presented libraries are discussed in some detail and their application is illustrated by examples.

1 Introduction

As automation systems are constantly increasing in complexity, new methods for controller design have to be applied. One promising approach is the concept of Distributed Automation Systems (DAS). Though distribution simplifies the design of complex control applications, analysis is more difficult than in traditional monolithic or strictly hierarchical systems.

Distributed systems have a concurrent nature. Hence, coordination and synchronization are needed between the individual control devices. This is usually achieved by means of networks. Because of the inexpensiveness of components, the plug-and-play abilities and the possibility for information access from higher level business units, standard networks like Ethernet currently tend to replace special purpose networks (ASI, ProfiBus, ...) in automation.

To design and analyze an automation system, the engineer relies on tool support. Individual tools for

algorithm analysis, network analysis and process simulation are available. However, the isolated analysis of any of these aspects does not meet the engineer's requirement of analyzing the closed-loop behavior of the system where the controller interacts with the controlled process via the network.

A simulation environment covering all aspects of distributed automation systems is Matlab/Simulink with the TrueTime [1] toolbox. Matlab/Simulink is a well-known tool for controller design and process simulation. TrueTime adds models for network and controller hardware. As Matlab/Simulink is used as platform, the analysis benefits from its advantages (widely spread, many process models available) but also inherits its disadvantages. Here, especially the causal procedural modeling approach in Matlab/Simulink complicates the design of complex process models and hinders the reuse of components.

Modelica, as a multi-domain modeling language, with the object oriented modeling paradigm and the non-signal-flow-dependant model causality, which increases the reusability of process models, is another adequate basis for overall system analysis.

The paper presents an approach for simulation of networks and controller hardware in Modelica in combination with process models, also modeled in Modelica. Means to analyze both, the functional and the temporal behavior of the overall system in early development stages are provided. First analysis results have already been presented in [2].

This paper focuses on the description of the developed libraries, including implementation details. In the next chapter, the used modeling objective is presented. Chapter 3 describes a library for simulation of network components. Together with the controller library, rendered in chapter 4, it is possible to model distributed automation systems. Both libraries have been implemented and tested with Dymola 6.1. Application examples of the presented libraries are given in chapter 5. Finally, conclusions are drawn and an outlook on future work on the libraries is given.

2 Modeling Objective

As in all simulation applications the initial step of the designer is to identify components and effects to be considered in the simulation. Based on this identification the next step is to decide about the modeling approach for implementation. In the case of distributed automation systems we have chosen a structure conserving modeling approach, mapping real world components to individual models. Figure 1 shows an archetype of a distributed automation system and the component models provided by the Modelica libraries presented in the following chapters.

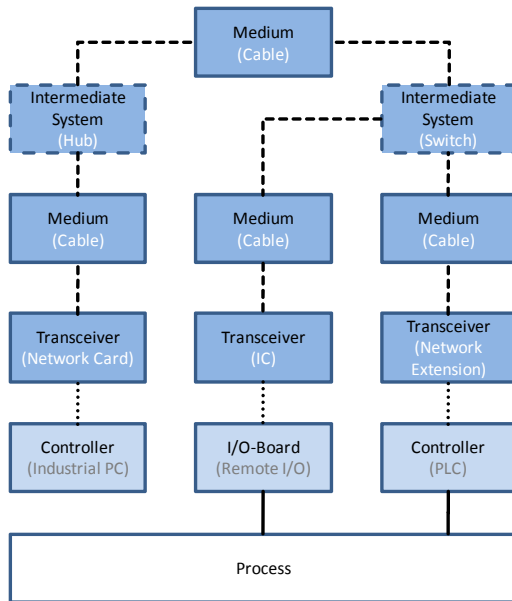


Figure 1: Archetype of systems that are to be covered by the presented libraries.

Basically, the components of the archetype can be divided in three domains:

The first domain (dark shading) covers the network. A network consists of one or more communication media (*Medium*). In the case of Ethernet this is usually a twisted pair cable. If more than one *Medium* is used, coupling devices (*Intermediate System*) have to be used. In Ethernet an *Intermediate System*, could be a hub or a switch. To access the *Medium* a *Transceiver* is used. It manages all physical and protocol issues necessary for proper communication over the network. The network card of a PC is an example of a *Transceiver*.

The second domain (light shading) covers components related to data handling and process interfacing. The components *Controller* and *I/O-Board* can be regarded as embedded devices. The control algorithms are executed in *Controllers* (e.g. industrial PCs or PLCs). In general, a *Controller* has access to the network via a *Transceiver* as well as access to the controlled process via directly connected sensors and actuators. *I/O-Boards* are a simplified version of a

Controller and allow remote access to sensors and actuators via the network. Usually, *I/O-Boards* have only limited processing abilities and are not used to execute control algorithms.

Components of the third domain (no shading) are related to the physical process.

There are three types of interconnections between components. Network connections (*dashed edges*) describe data wrapped in a protocol frame, dependant on the network type used. Pure data transport is indicated by *dotted edges*. The exchange of physical values is shown as *solid edges*.

Based on the domain classification of components in distributed automation systems, in chapter 3, a Modelica library for network components is presented. The library described in chapter 4 covers the components related to embedded devices for process control.

Both libraries make intensive use of the object orientation abilities of the Modelica language. Interface models are used to allow the exchange of components with similar behavior. Wherever possible and appropriate, components implement a predefined interface or are extended from other existing models. Along with the Modelica keyword *replaceable*, this allows e.g. a wide variety of controller models based on a small number of basic components. In the figures of component models replaceable component models can be identified by the gray shaded box around them (e.g. Figure 10 component CPU).

3 Network Library

3.1 Structure of the library

The network library consists of fundamental components which cover the important issues in the area of network transmission, e.g. communication media, intermediate systems, transceiver interfaces, etc. The rule of structure conserving modeling is held. I.e., the network is not modeled as a single class but all the fundamental components are explicitly modeled. The main advantage of this modeling approach is that the network topology, which can have significant influences on the network performance, is visible.

Currently, the library (Figure 2) supports three widespread transmission protocols, namely, fully switched Ethernet [3], WLAN [4] and ZigBee [5]. To increase simulation speed, the protocols are simplified to some extent, thus, only the chosen dominant factors related to the automation system are modeled. Especially noteworthy is the fact that in the network components, only the physical and data link layers of the ISO/OSI model are considered and also here some abstractions have been made. Necessary interfaces for

the exchange of events are also included. In the *Functions* library, *external* “C” functions and corresponding wrap functions in Modelica can be found. External functions are used to simplify information exchange, especially for the formatted string communication. In the *Examples* library, application templates are given for each protocol, illustrating how to build a networked system using the models.

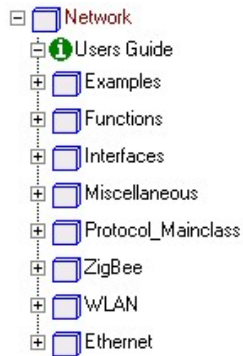


Figure 2: Tree view of the network library.

3.2 Application template of the network library

Figure 3 shows a typical application example of a ZigBee network. The dark shading blocks on the left-most side represent primary controller models. They provide the data to be sent and read the data from network messages regardless of the underlying transmission protocols. In the middle there are two transceiver modules connected to the controller and the shared medium. The transceiver module together with the medium defines the transmission protocol. In the example, controller ‘A’ sends messages to controller ‘B’ via the network while ‘B’ does the same to ‘A’.

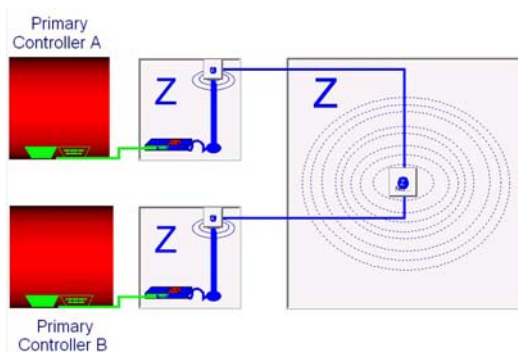


Figure 3: Application template of a ZigBee network.

The source controller renews the data to be sent cyclically and writes the message into the send queue of the transceiver module. Based on the network state and medium access algorithm, the transceiver module decides when to send the network message. After the transmission duration has expired, the network medium writes the message into the receive queue of the destination transceiver. Then, the destination control-

ler determines when to read the data from the network message stored in the receiver queue.

3.3 Implementation of the queuing system

As noted above, the transceiver module serves as an interface between the controller and the network. In addition to the medium access control, it has to accomplish the information exchange. A transceiver interacts with both, the controller and the medium (cf. Figure 4).

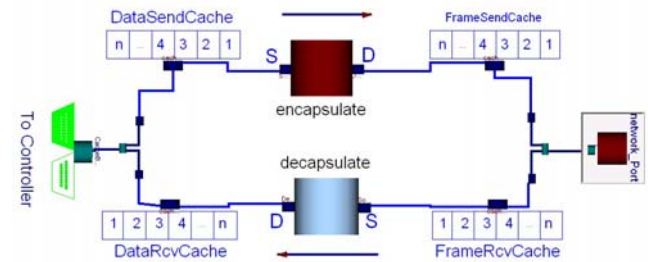


Figure 4: Queuing mechanisms in the Transceiver.

Thus the information flow in a transceiver is bidirectional. Furthermore, the information flow in one direction is split into two segments due to the unsynchronized behaviors of controller, transceiver and network. To manage the information flow, FIFO queues are utilized. Each information flow segment is represented as a FIFO queue. The two nodes on both ends of the same information flow share the access to the same queue. The FIFO queue system is implemented as *external* “C” functions to simplify the Modelica code and reduce the number of events. The interface functions in Modelica are given as:

```
QueueID=CreateQueue(QueueSize);
Enqueue(QueueID,MessageID);
MessageID=Dequeue(QueueID);
Index=ReadQueueIndex(QueueID);
```

CreateQueue(QueueSize) creates a queue with given length and returns a unique ID. Enqueue(QueueID, MessageID) stores the MessageID in the first free place of the queue. Dequeue(QueueID) reads the first message from the queue and shifts the rest of the queue one place towards the beginning position. ReadQueueIndex(QueueID) returns the current position index. The entries of a queue are message identifiers. Each message (string) is indexed with a unique integer ID. This utility is supported by a “C++” library.

Figure 5 illustrates the information exchange in queues for the example from Section 3.2. New data (string) from the controller is represented by an identifier (ID1). This ID is enqueued in *DataSendCache*. In the next step, ID1 is dequeued and the actual content of ID1 is encapsulated to a frame with protocol header. Thus, a new message (string) is produced and a new identifier (ID2) is enqueued in *FrameSendCache*. In the transmission, medium dequeues the

ID2 and makes a copy named ID3. Later after the transmission, ID3 is enqueued in *FrameRcvCache* on the destination side. Finally, ID3 is decapsulated and a message ID4 is enqueued in *DataRcvCache*. During the whole procedure, each time the *Dequeue()* function is called, a copy of the dequeued message is made and the original message is deleted. Hence, the message ID4 and ID1 actually have the same content. The information exchange is accomplished.

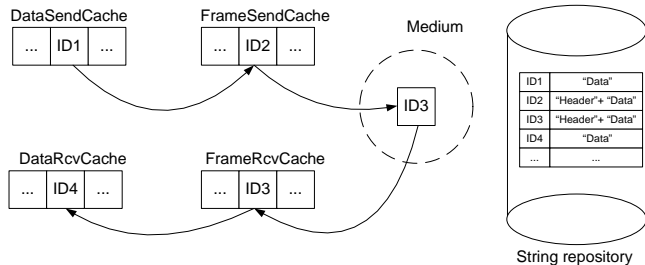


Figure 5: Queue operation in a transmission.

3.4 Implementation of Ethernet

The implemented Ethernet protocol is abstracted from a fully switched, full-duplex Ethernet. The network behavior is illustrated in Figure 6.

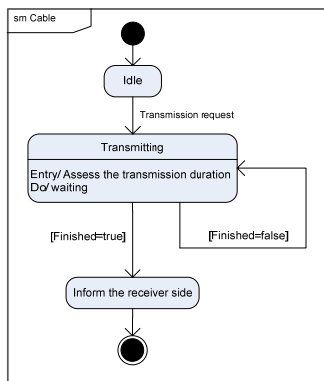


Figure 6: State diagram of Ethernet protocol.

A twisted-pair cable is taken as the modeling pattern for a medium in Ethernet. Since no collisions are considered, the end of a cable can only be connected to one transceiver. In a simulation, the cable model receives request events from connectors on both ends and sends out notifications on connectors after the internal processing.

3.5 Implementation of wireless communication

In principle, the network is modeled as a discrete event system. It reacts on external events with deterministic or non-deterministic delays. The modeling focuses on the standard MAC layer taking into account random access and conflict handling.

The implementation of the protocol is divided into software and hardware parts. The software part covers medium access, frame format etc. It is the algorithm integrated in the transceiver and intermediate

system. The hardware part is the communication medium. The communication state, the transmission duration and other relevant variables are decided by electromagnetic characteristics of the medium. From this point of view, there is always a clear relation between the network protocol and the underlying physical medium. In the presented approach, parts of the protocol codes from the transceiver model are moved into the medium model. In other words, the medium is designed with some extent of intelligence.

In wireless communication, it is not possible to listen while sending because of the nature of the channel (frequency band). Hence, the Collision Avoidance (CA) method is used to improve the performance of Carrier Sense Multiple Access (CSMA). In principle, a network node always listens to the channel and sends only if the channel is sensed as idle. The implementation of this protocol is separated into two parts, namely, the medium part and the transceiver part. The interaction between medium and transceiver is realized by *Network_Port* which can be found in *Network.Interfaces*.

There are two main differences between the CSMA/CA algorithms for ZigBee and WLAN on the MAC layer:

1. WLAN has an unlimited number of retries, while ZigBee is strictly limited on retries.
2. ZigBee assesses the network state only at the end of the whole backoff time, while WLAN checks after each single delay unit.

Then again, they do have some important characteristics in common, e.g. listening before sending, random backoff waiting time before sending, incremental backoff time after collision.

Therefore, the modeling attempt is to design a common model for these two algorithms. The differences can be represented by changing model parameters. The implementation is based on the unslotted CSMA/CA scheme, which means the network works without beacon synchronization and all nodes are working in the Ad-hoc mode. Thus the access to the network is random and contention prone. Details about the protocols can be found in [4] and [5].

The common wireless medium model is illustrated in Figure 7. The shared medium is triggered if any network node sends an attempt of trying. Based on the sum of trying nodes, it decides to begin transmitting or to send a collision notification. After successful transmission, it waits for a certain time before resetting the medium state to idle. This time is given by:

$$WaitingTime = SIFS + ACK + DIFS \quad (1)$$

where SIFS (Short Inter Frame Space), ACK (Acknowledgement), DIFS (Distributed Inter Frame Space) are physic dependent parameters defined in the standard.

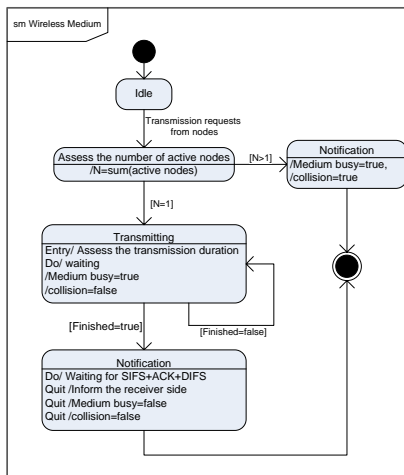


Figure 7: State diagram of wireless medium.

The transceiver part algorithm is mainly used to perform the backoff procedure. Here, the separate implementation shows great advantage in the WLAN protocol considering the simulation performance. Since the conflict detection is executed by the medium model, the transceiver models do not have to assess the medium state after each unit delay, but only to wait for the event trigger from medium notification. By doing so, unnecessary events, which slow-down the simulation dramatically, are avoided. For instance, 802.11 standard defines the backoff time in the unit of timeslots:

$$\text{Backofftime} = \text{backoff_counter} \times \text{slot_time} \quad (2)$$

where the initial backoff_counter is a random number in the range [0, 31] and slot_time is 20 μs [6]. If no collision happens, it causes on an average 16 events with a cycle of 20 μs per transmission. In the worst case (4 collisions happen successively), there are 1031 events for a transmission. In the separate implementation, the number of events is reduced to 1.

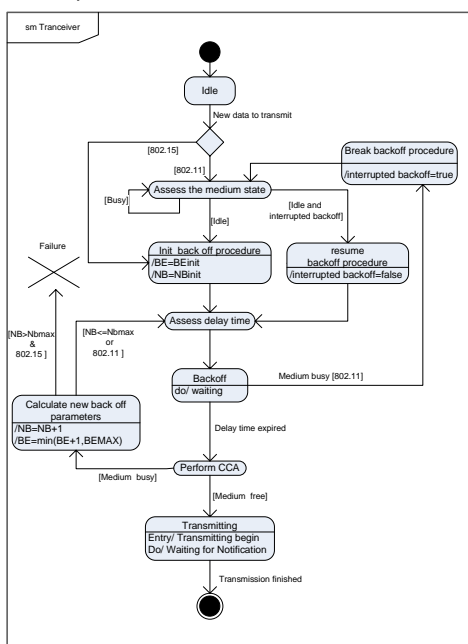


Figure 8: State diagram of wireless transceiver.

The designed common model is illustrated in Figure 8. The exact model behavior is predefined by a model parameter given as “802.11” or “802.15”. The common model is a partial model in Modelica, thus in the application, it is instantiated as a replaceable model and can be easily parameterized for different protocols.

There are some important assumptions to be noticed:

1. No transmission failure is taken into consideration, i.e. no packet is lost in the transmission and no re-transmission is needed.
2. The acknowledge message is not modeled.
3. One shared medium model represents one available channel. All nodes connected to the medium hence operate in the same channel. No dynamic channel switching is considered. As a consequence, the network capacity is restricted.

4 Controller components library

4.1 Overview of the library

The controller components library contains models to describe the behavior of an embedded controller device. In comparison to the simulations of automation systems without detailed controller models, the effects of synchronization, scheduling and queuing are considered in retrieving system behavior which provides more realistic simulation results.

The library is split into sub-libraries that group controller components by their function (cf. Figure 9).

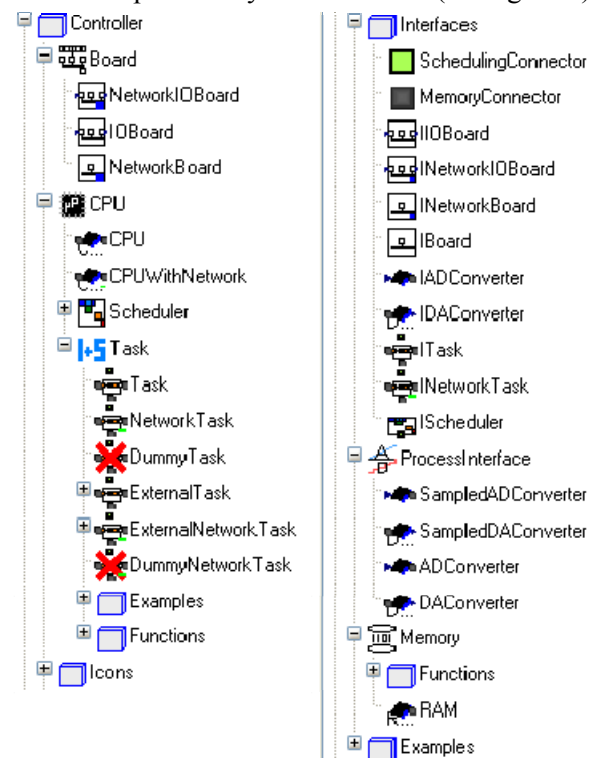


Figure 9: Screenshot of the controller library.

Board

The basic component of a controller device is a board. It hosts devices that are needed to run control algorithms, to interface plants to be controlled and to interchange information between controller devices. The library *Controller.Board* hosts three basic board models which distinct in the interfaces they provide (only process interface (IOBoard), only network interface (NetworkBoard), combination of both(NetworkIOBoard)). The components of the NetworkIOBoard model, shown in Figure 10, will be detailed in the following sections.

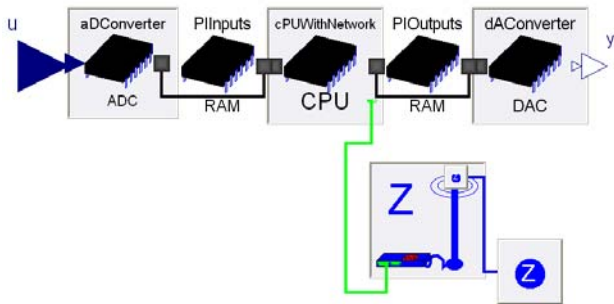


Figure 10: Component model of a board with process and network interfaces.

Process Interfaces and Converters

The process interface consists of an array of continuous input (u) and output (y) signals. As in real controllers input and output signals are not directly connected to the CPU. The input signals are first converted by a hardware AD-Converter, and the DA-conversion of the output signals is done by means of a hardware DA-Converter. The converter models can be found in the *Controller.ProcessInterface* library.

RAM

The results of the AD-conversions are stored in a random access memory (RAM) called process image of inputs (PIInputs), whereas the output signals to be DA-converted are read from the process image of outputs (PIOOutputs) by the DA-Converter.

The RAM model can be found in *Controller.Memory*. It provides means to exchange information between component models. To attach models to a memory component the *MemoryConnector* is used.

Network Interface

The network interface is provided by the network library. It is regarded as a transceiver IC which performs network operations concurrently. The CPU can transfer messages to be sent via network to the transceiver IC whereas received network messages can be read by the CPU from the transceiver IC.

CPU

The central processing unit (CPU) executes the control algorithms. The CPU models can be found in the library *Controller.CPU*. There are two different

CPU-models in the library, one without network access (*CPU*) and one with network access (*CPU-WithNetwork*). The *CPU* model is instantiated as replaceable in the board models and, thus can be exchanged to other *CPU* models extending *CPU*. The CPU executes the control algorithms wrapped in Tasks as described in the next section. Figure 11 shows the *CPUWithNetwork* model.

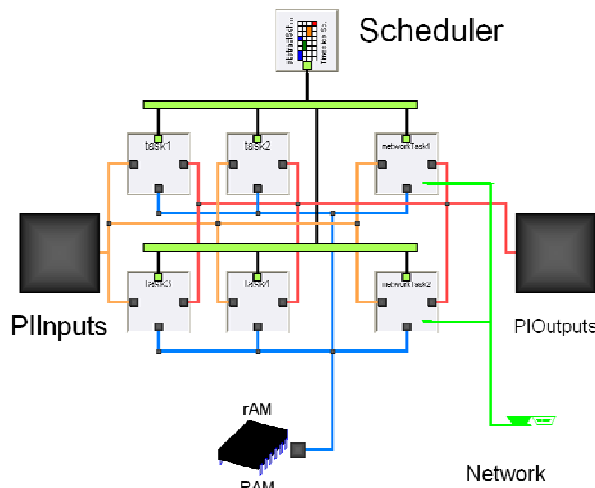


Figure 11: Model of a CPU with network access.

Task and Scheduler

A task (library *Controller.CPU.Task*) is a software process which runs quasi-concurrently to other tasks in a CPU. A scheduler (library *Controller.CPU.Scheduler*) allocates processing time to the tasks according to a certain scheduling policy. The library provides several scheduler models (e.g. Round Robin, FIFO) which are all based on the *Controller.Interfaces.IScheduler* model.

All tasks have access to the process images (PIInputs and PIOOutputs) and share a common memory called RAM. The RAM can be used to exchange information between tasks inside a CPU. The connections to the different memory types are drawn in different colors in Figure 11 (PIInputs: orange, PIOOutput: red, RAM: blue). The connections between the tasks and the scheduling model are drawn in black. The green horizontal bars are used as a Modelica bus to reduce connections to the scheduler.

The *CPUWithNetwork* model can host up to four tasks without network access (*Task*) and two tasks with network access (*NetworkTask*). *Task* and *NetworkTask* are implemented as partial models and serve as generalized task models. The task models are instantiated as replaceable in the *CPUWithNetwork* model and can be easily changed to specialized ones. The models *DummyTask* and *DummyNetworkTask* can be used to specify that a task is not present. The library user can easily build his own *CPU* models with more Tasks by extending the existing ones.

4.2 Implementation details

RAM

The *Controller.Memory.RAM* model plays an important role in the simulation of an embedded controller. It provides means to interchange information between controller components. On the first look, this does not seem to be a big issue in Modelica, as the *connector* type is especially designed for this purpose. But, in the domain of informational systems a connector is not a convenient tool to exchange information between components. Due to the fact that the amount of data exchanged may vary, not all cases can be covered when designing a general information exchange connector. Instead, it is appropriate to make use of the external function interface of Modelica to implement an information exchange system in a dedicated programming language which is then triggered by Modelica models. In this way the complexity of data handling is hidden from the Modelica models. To do so, a C++ library has been developed which emulates a collection of random access memories that can be accessed via a unique index. Figure 12 shows a coarse overview of the C++ library internals and the interconnection with the Modelica model.

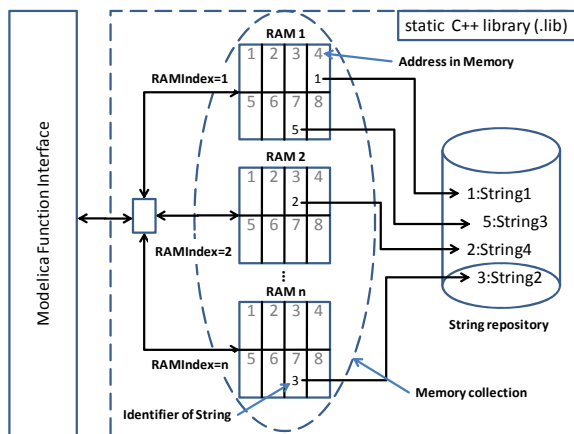


Figure 12: Coarse overview of the C++ library.

Interface functions to use the C++ memory collection are provided in the sub-libraries *Controller.Memory.Functions*. *WrapperFunctions*:

```
RAMIndex:=createRAM();
value:=readRAM(RAMIndex, address);
writeRAM(RAMIndex, address, value);
```

The function *createRAM* creates a random access memory in the collection and returns the unique identifier of the memory (*RAMIndex*) as an Integer. When reading or writing from or to a memory, the *RAMIndex* has to be passed to identify the memory.

readRAM is used to read values (as String) from the memory *RAMIndex* from the given position *address* (as Integer) in the memory, whereas *writeRAM* is used to store a value in the given memory *RAMIndex* at position *address*.

In the C++ library, a RAM is organized as a collection of references to strings (character arrays) stored in a global string repository. Library internal functions provide the mechanism to read and write strings to the string repository, identified by the *RAMIndex*, using the given *address* which is a local identifier inside a RAM.

A slightly varied functionality is used to implement the queuing mechanism in the same library. Instead of giving random access to values in a *queue*, the interface functions only allow reading (dequeuing) from the first address in the queue and writing (enqueueing) at the end of the queue. The string management is done in the same way as in the case of RAM, using the string repository.

For maximum flexibility, the values passed to or from the memory are of type String. This way arbitrary information can be used in the simulation. When necessary, numerical information contained in the strings can be parsed by means of functions provided in the *Modelica.Utilities.Strings* library. In the case of real values, the library provides the functions *readReal* and *writeReal* in the sub-library *Controller.Memory.Functions*, where parsing is done automatically.

The RAM model itself is just a placeholder for one of the memories managed in the C++ library. Its only dynamic behavior is a function call to create a memory in the memory collection in the initial simulation step. The unique index of this memory is then stored in the RAM model and published via the *MemoryConnector*, which only consists of the unique memory index.

To improve debug capabilities, the interface functions provide the possibility to trace read and write accesses to the RAM model. For this purpose, each function call, including function parameters, of *readRAM* and *writeRAM* can be stored in a textfile or a database table (cf. Table 1).

Table 1: Dump of a database trace of operations on RAM models.

id	operation	RAMindex	address	value
		//		
28	W	1	2	"5"
29	W	2	5	"-3"
30	R	1	2	"5"
31	W	2	3	"3"
32	R	2	2	"1"
33	R	1	2	"5"
		//		

Tasks

As described above, the control algorithms are organized in tasks to allow multiple algorithms to run quasi-concurrently on a single CPU. Hence, if multiple tasks are active at once, they are competing for

processing time. The scheduling instance manages that the processing time is spread among all tasks.

The task models (*Controller.Task.**) can be seen as wrapping units that provide interfaces to access CPU-internal (e.g. Memory, Timer) and CPU-external components (e.g. process images, network port). Therefore, all tasks extend the *Controller.Interfaces.ITask* interface. The behavior is described in Figure 13.

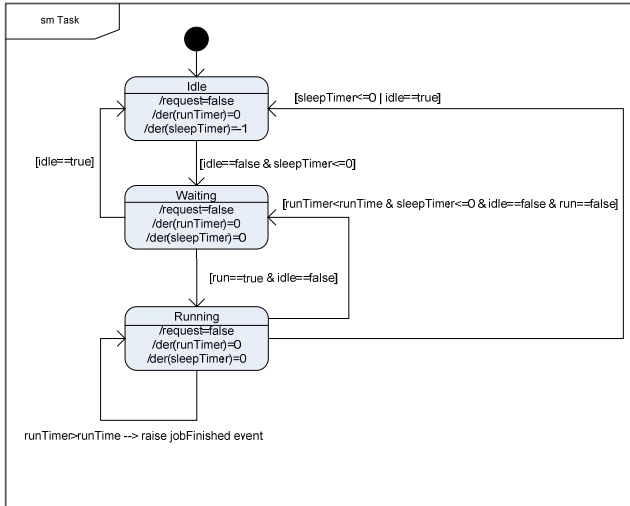


Figure 13: State diagram of a task.

A task can be in one of three states (*Idle*, *Waiting*, *Running*), and is controlled by four variables: *run*, *idle*, *runTime* and *sleepTime* which enforce state switches. Depending on the state, the timers *runTimer* and *sleepTimer* are active or inactive and processing time is requested or not (variable *request*).

The external variable *run* is provided by the scheduler and determines if the task is currently the active task in the CPU. The internal variable *idle* enforces the task to go to *Idle* state if its value is true. *runTime* determines the processing time that is needed to finish the current job of the task, while *sleepTime* is used to assign the period to send the task to *Idle* state.

If a task has no job to do, it is in *Idle* state. In state *Waiting*, the task is requesting processing time, but currently does not get processing time by the scheduler. In state *Running*, the task gets processing time and is thus running. When the *runTimer* value exceeds *runTime* the *jobFinished* event is raised which indicates that the current job of the task is completed. This event is used to embed the actual control algorithm in the *Task* model, just by extending one of the given template tasks (*Task* or *NetworkTask*). The behavior is given by the extended *Task* model. This way the library user can focus on the implementation of the algorithms itself.

Integration of Algorithms

The Controller library provides basically two different ways to define the algorithms executed in a *Task* model:

1. algorithm definition in Modelica language
2. algorithm definition in external libraries using the Modelica external function interface

To define and implement an algorithm in Modelica, one of the basic *Task* models (*Task* or *NetworkTask*) must be extended. As intelligence for scheduling is predefined in the basic *Task* models, only the values of the task control variables *idle*, *sleepTime* and *runTime* as well as the reaction on the *jobFinished* event must be specified.

The basic structure of tasks using Modelica algorithms is defined as shown below.

```

model ExampleTask
  extends Controller.CPU.Task.Task;
  initial algorithm
    sleepTime:=0;
    runTime:=100/CPUFreq;
  algorithm
    when (pre(jobFinished)) then
      //here, the algorithm semantics are specified
    end when;
end ExampleTask;

```

The *initial algorithm* section assigns start values for the control variables *sleepTime* and *runTime*. In the *algorithm* section, the processing of the *jobFinished* event is defined by means of a *when* block. The condition *jobFinished* must be wrapped by a *pre* to cut the algorithmic loop involving the task and the scheduler component.

The semantics of the algorithm is then specified in the body of the *when* block, as shown in the following example of a single input single output P controller:

```

...
import Functions=Controller.Memory.Functions;
...
when (pre(jobFinished)) then
  //when a job is finished, runTimer is reset
  reinit(runTimer,0);
  //state==0: read input value
  if (state==0) then
    //read sensor value from process image
    y:=Functions.readReal(PIInputs, 1);
    //next state is executing control law
    state:=1;
    //executing control law takes 5000 cycles
    runTime:=5000/CPUFreq;
    //state==1: execute control law
  elseif (state==1) then
    //calculate control error and new set value
    e:=w - y;
    u:=k*e;
    //next state is writing new set value
    state:=2;
    //execution for writing takes 500 cycles
    runTime:=500/CPUFreq;
    //state==2: set output value
  elseif (state==2) then
    //write output value to process image
    Functions.writeReal(PIOutputs,1,u);
    //next state is reading input values
    state:=0;
    //reading input signal takes 500 cycles
    runTime:=500/CPUFreq;
  end if;
end when;
...

```


The task is divided into three jobs which are executed sequentially. The job currently processed is indicated by the Integer variable *state* (0: read inputs, 1: calculate output, 2: write output). The execution time can vary among the jobs and is expressed in terms of processor cycles divided by the processor frequency (*CPUFreq* in Hz). When the *jobFinished* event arises, the semantics of the job is being performed and the settings for the next job are carried out. This means that the *runTimer* variable is reset to zero and that the *runTime* for the next job is assigned. The task should run without breaks and thus is never sent to *Idle* state (*sleepTime:=0, idle:=false*).

As described earlier, the tasks do not have direct access to the environment (input and output signals). Instead, the algorithms work on images of the input and output signals by reading from the process image of inputs (*PIInputs*) or writing to the process image of outputs (*PIOutputs*) using the provided interface functions.

For complicated algorithms, the Modelica design language is not the mean of choice. For this case, the *Controller.CPU.Tasks* library provides the *ExternalTask* and *ExternalNetworkTask* models, which allow the definition of algorithms in other programming languages (e.g. C or Java). Using the construct of *replaceable function*, the user of the library can easily access external algorithms by providing Modelica interface functions extending *Controller.CPU.Tasks.externalAlgorithm*.

5 Application of the Libraries

Many analysis problems in DAS can be characterized as runtime or delay investigations. A typical real world problem is the determination of the response time distribution on events in the process to be controlled (e.g. emergency stop). Results, elaborated with an earlier version of the presented library, have been published in [7].

Including the plant under control in the simulation, quality analysis can be performed. Reaction delays, due to the distributed nature of DASs, cause fluctuations in the control quality. As shown in [2], the libraries can be used to analyze the variations of process values based on a collection of simulation runs.

Another application of overall system simulation is feasibility analysis. It can be used to perform proof-of-concept tests in early development stages of DASs. The classical example for feasibility analysis in continuous control is the stabilization of an inverted pendulum in the instable (upper) rest position.

Figure 14 shows the setup of a DAS with an inverted pendulum using a wireless network. The experimental setup consists of the inverted pendulum with

0.5 kg mass for the cart as well as for the pendulum arm with a length of 1 m. In the initial, state pendulum arm and cart are not moving, but the pendulum arm is rotated by $\varphi = 0.25$ rad.

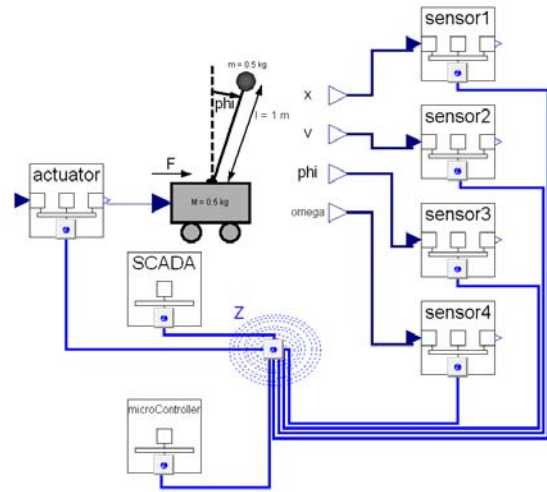


Figure 14: Inverted pendulum experiment.

Attached to the pendulum there are four *Controller.Board.IOBoards* (sensor1-4) providing information about position and velocity of the cart as well as angle and angular velocity of the pendulum arm. The *Controller.Board.IOBoard* actuator drives the cart with the translation force calculated by the *Controller.Board.NetworkIOBoard* microController. The *Controller.Board.NetworkIOBoard* SCADA (Supervisory Control And Data Acquisition) is allotted to collect the overall system status periodically.

The control algorithm on the microcontroller is split up into three concurrent tasks. The first task cyclically requests sensor values from the sensor boards and sends the calculated force value to the actuator board. The second task handles incoming network messages, and is only active when network messages are available. The third task is the control algorithm itself working in three sequential steps:

1. reading input signals from the RAM, provided by the remote sensors,
2. calculating the force value (control law),
3. writing the force value to the RAM.

The control law is a state controller, with gains acquired from a linear continuous time state space model.

The requesting task runs 4 ms and then falls asleep for 20 ms. The message handling task needs 1 ms to process each incoming message. The control algorithm task needs 1 ms to read the input signal values as well as 1 ms to write the force value. To execute the control law, 11 ms are needed.

Processing of network messages in the sensor and actuator boards takes 2 ms. As there are no concurrent tasks in these boards execution starts as soon as a network message is received.

The example setup has been simulated with two different networks, ZigBee with 250 kbps transmission rate and WLAN with 11 Mbps transmission rate. Additionally, the inverted pendulum has been simulated using the same control law emulating a continuous controller neglecting all delays imposed by the automation system.

As shown in Figure 15, the pendulum angle varies a lot among the three simulated scenarios. As expected, the scenario neglecting delays shows best performance and the WLAN scenario is superior to the ZigBee scenario.

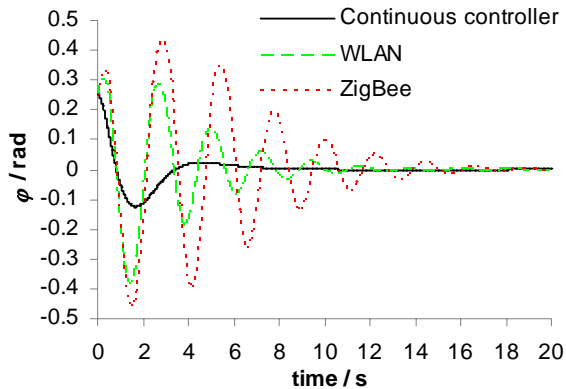


Figure 15: Pendulum arm angle sequence plot.

The reason for the differences among the scenarios is the delayed application of the actuator force caused by the automation system delays. It can be seen from Figure 16 that the first update of the actuator value occurs after approx. 80 ms using WLAN. In the scenario with ZigBee this delay increases to 100 ms.

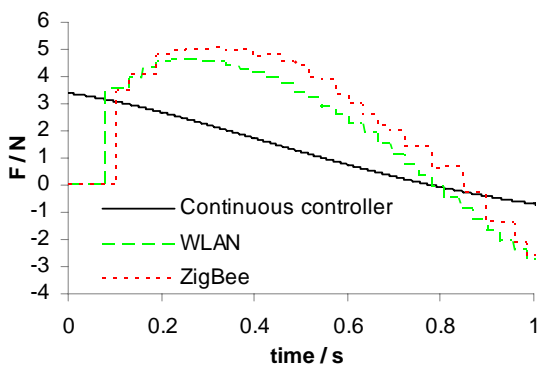


Figure 16: Zoomed applied translational force sequence plot.

A 20 s lasting simulation (carried out in Dymola 6.1) takes 94.2 s CPU time on a PC with 2.8 GHz Pentium IV HT and 2 GB RAM. Best performance has been achieved using the *Lsodar* integration algorithm with $1e-6$ tolerance.

Using the common setup for storing variables consumes too much memory and disk space, even for short simulations (less than 10 seconds). Thus, interesting simulation values have to be stored using pro-

prietary mechanism, e.g. sampled saving of data to a file with using *Modelica.Utilities.Streams.print*.

6 Conclusions and Outlook

Libraries for simulation of Distributed Automation Systems using the Modelica language have been presented. The libraries allow delay time determination, quality of control analysis and feasibility analysis in closed-loop applications. The application of the libraries has been illustrated by an example using wireless communication.

Future work will focus on improvement of the network models regarding failure behavior (e.g. packet losses) and the integration of other networks especially in the field of automotive applications (CAN, LIN, ...).

The presented libraries can be downloaded from <http://www.eit.uni-kl.de/frey>.

References

- [1] Cervin, A.; Ohlin, M.; Henriksson, D. *Simulation of networked control system using TrueTime*. 3rd International Workshop on Networked Control Systems: Tolerant to Faults, Nancy, France, June 2007
- [2] Liu, L.; Frey, G. *Simulation Approach for Evaluating Response Times in Networked Automation Systems*. IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA), Patras, pp. 1061-1068, Sept. 2007.
- [3] IEEE Std 802.3, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 1999.
- [4] IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999.
- [5] IEEE Std 802.15.4, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) 2003.
- [6] Ferre, P.; Doufexi, A.; Nix, A.; Bull, D. *Throughput analysis of IEEE 802.11 and IEEE 802.11e MAC*. Wireless Communications and Networking Conference (WCNC), Vol. 2, pp. 783 - 788 March 2004
- [7] Greifeneder, J.; Liu, L.; Frey, G. *Methoden zur Antwortzeitanalyse in vernetzten Automatisierungssystemen*. SPS/IPC/DRIVES, Nürnberg, Germany, pp. 517 - 525, Nov. 2007.

Session 2a

Language, Tools and Algorithms

Study of a sizing methodology and a Modelica code generator for the bond graph tool MS1

†Audrey Jardin, †Wilfrid Marquis-Favre, †Daniel Thomasset, *Franck Guillemard, †Francis Lorenz

†AMPERE
INSA-Lyon
25, avenue Jean Capelle
F-69621 Villeurbanne Cedex

*PSA Peugeot Citroën
Centre technique de Vélizy
Route de Gisy
F-78943 Vélizy-Villacoublay Cedex

†LorSim
89, rue Jacob-Makoy
B-4000 Liège

firstname.lastname@insa-lyon.fr
franck.guillemard@mpsa.com
francis.lorenz@lorsim.be

Abstract

Complex systems engineering requires new software tools for virtual prototyping which have to be more relevant in order to meet, at the same time, consumer requirements, standardized rules and market law. These have to be more flexible especially concerning file exchange and reusability. Recently the modelling language Modelica seems to fulfill these needs thanks to its concepts of acausality and multi-disciplinary description.

In parallel, the laboratory AMPERE has developed a bond graph-based sizing methodology which, by the use of inverse models, drastically decreases the number of calculus iterations compared to the classical direct approach.

The aim of this paper is to highlight the importance of acausality and structural analysis in a design approach and to study to what extent the proposed sizing methodology can be formulated in Modelica. Then first software implementations of the methodology are illustrated by examples processed by the tool MS1 and its Modelica code generator.

Keywords: code generator; Modelica; MS1; bond graph; acausality; structural analysis; sizing methodology

1 Introduction

Nowadays technological advances have lead to systems which are more and more complex and thus, more and more difficult to design. In the new context of sustainable development, systems have to match

ever-increasing pollution standards while engineers have to take into account both higher consumer requirements (like safety, comfort, equipment,...) and financial constraints. In few words, engineers have to conceive faster new safer and cheaper solutions.

One way of doing that is to proceed by simulation which has the benefit to avoid costly manufactures of several impertinent prototypes and then favour gain of time and money.

However virtual prototyping is really efficient only if the engineer is able to accurately model the system, i.e. only if the system is sufficiently described for the given problem. In fact, the hardest tasks of such an approach are:

- finding the good description level;
- being able to express the different physical phenomena implied by this description;
- and representing these in an unified manner even if they involve various physical domains.

For all of these reasons, engineers need a modelling language which:

- allows making connection between all kinds of physical domains.
→ The modelling language has to be multi-domain.
- ensures a sort of continuity at every level of the project cycle. So models have to be usable as well in oriented system softwares during a pre-sizing phase as in more specialized tools in advanced design steps.
→ The modelling language has to be recognized as a standard for model exchange.

- reduces wasted time as much as possible. In fact, it is of the first importance to mutualize modelling efforts which, as mentioned before, are the hardest tasks of such an approach. One way of capitalizing on this is to separate the system description from the design context and thus not to depict the system with *a priori* oriented equations.

→ The modelling language has to be object oriented and to enable acausal description.

- reduces study costs by decreasing dependency towards exclusive software providers.

→ The modelling language has to be a free and non-owner language.

This is just with this in view that the Modelica language and the OpenModelica simulation environment [1] have been proposed. In fact this can explain why, today, Modelica language seems to fulfill a real need for engineers and industrials and seems to present itself as the future standard for model exchange. As a proof of fact, numerous simulation environments and computer aided design tools like Dymola [2], LMS Imagine.Lab AMESim [3] or Scilab/Scicos [4] can now support Modelica models as well for import as for export.

Starting from this statement, the aim of this paper is to compare some Modelica aspects to the bond graph-based sizing methodology [5]-[12] developed by the laboratory AMPERE¹. In fact, by using the multi-domain aspect as well as the concept of acausality, it seems legitimate to ask oneself to what extent the proposed methodology can be supported by Modelica language.

The paper is organized as follows. First, section 2 will briefly describe the methodology principles and its benefits compared to a classical design approach. Importance of the acausality concept and the use of a structural analysis will also be highlighted. Section 3 will present one example of the methodology software implementation, the tool MS1 [13], and its newest functionality: a Modelica code generator. Then section 4 will conclude by summarizing the several tackled points and by suggesting future research directions.

¹ Since January 1, 2007, the LAI has merged with the CEGELY and a team of environmental microbiology to become the laboratory AMPERE (UMR CNRS 5005).

2 Bond graph-based sizing methodology towards a Modelica-based sizing methodology?

To understand how some Modelica features can be used or be augmented to support the proposed methodology, it is worth first explaining its main principles. Then importance of an unified and acausal description will prove to be a benefit for carrying out a structural analysis. Finally some reflections will be conducted about the potential of embedding the methodology in Modelica.

2.1 Methodology benefits and principles

Up to now a classical approach adopted by the most of engineering departments consists of a trial and error procedure. For instance consider an actuated load system (Fig. 1) and suppose that the design problem is to find an appropriate actuator so that the load follows a given trajectory (i.e. the hoped-for specification). Once the system has been modelled, the first step of a classical approach consists in:

- selecting more or less arbitrarily an actuator (this depends on the degree of the engineer expertise);
- presupposing the control of this actuator;
- launching a direct calculus in simulation according to these assumptions;
- comparing the calculated load trajectory to the desired specification.

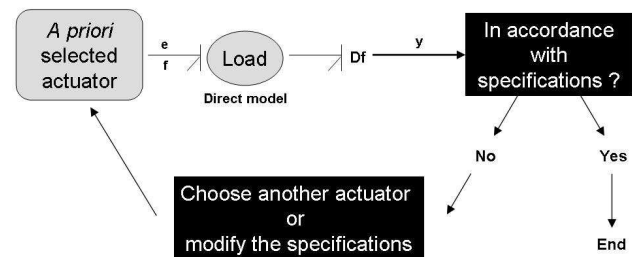


Fig. 1: The classical design approach

However this approach rarely leads to a good solution at the first attempt: it usually requires numerous iterations to find a suitable actuator. This is truer in a technological break context where, by definition, engineers do not have access to any expertise. Moreover this approach can come up to a greater loss of time since:

- in the first case where the *a priori* selected actuator matches the specifications, the engineer has no idea on the margins he has at his dis-

posal, and thus whether a smaller and cheaper actuator could be acceptable;

- in the second case where the *a priori* selected actuator does not suit the sizing problem, the result of the simulation does not give any idea on the causes of underdimensioning. The engineer in charge of the study must choose another actuator admittedly more powerful but still more or less arbitrarily.

Finally this iterative procedure can even reveal itself endless as, beforehand, no checking has been made to conclude whether the specifications can be really obtainable by the given structure or not. In that case, most of time, the engineer has to slightly modify the specifications by relaxing some design constraints if he wants to solve his problem.

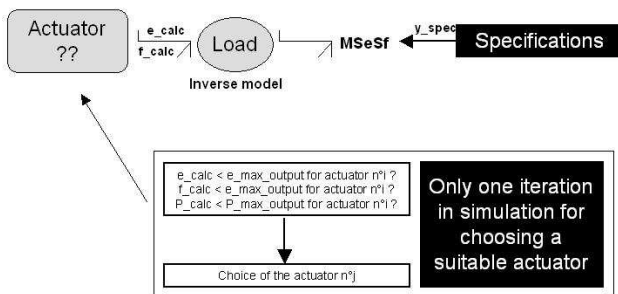


Fig. 2: The laboratory AMPERE design approach for choosing suitable actuators

Faced to all of these drawbacks and strong of its research for 15 years, the laboratory AMPERE has developed an innovative methodology for sizing mechatronic systems ([5]-[8]). Contrary to the classical approach which uses direct model calculus, the key idea here is to exploit inverse models described by bond graph. Considering the same example as before, the main steps of this approach can be summed up into the following points (Fig. 2):

- **Step 1: Adequacy**
As explained in more details in section 2.3, this step consists in carrying out a structural analysis. This allows checking if the sizing problem is well-posed and concluding on the possible structural invertibility of the load model (and so on the possibility to inverse the model).
- **Step 2: Specification**
Assuming that the load model is structurally invertible, this step consists in establishing the inverse load model corresponding to the given sizing problem (this results in assigning the bi-causality on the bond graph model) and simu-

lating it so as to determine variables required at the entrance of the load and that match the specifications².

- **Step 3: Selection**

As the variables in input of the load are the same as the variables in output of the actuator, the engineer can thus select in a library actuators that appear suitable for the output specifications (e.g. the maximum of the required effort must be inferior to the maximum effort the actuator can supply) (Fig. 2).

- **Step 4: Validation**

Finally since actuators have been selected according to criteria only in terms of variables in output, the engineer has to check if these actuators do not overcome their limitations in input (and anywhere else in the inside). This step consists in adding the actuator models to the load model, determining the variables in input by the use of the new corresponding inverse models³ and comparing the simulation results for these variables to the manufacturer data.

Then these four steps of the methodology are repeated to size each stage of a whole actuating chain (power modulator, energy supplier) and, at the end, to determine the open loop control.

Now that the principles of the methodology have been exposed, it is worth noting some remarks.

First, the methodology does not require any supposition on the actuator control and, by this way, facilitates the engineer study.

Secondly, compared to the classical approach, the inverse methodology drastically decreases the number of calculus iterations. In fact, at the end of the selection step, as the variables needed in output of the actuator are directly determined from the specifications, the engineer is able, after only one calculus, to:

- either eliminate a whole part of the actuator library (whereas each component should have been tested in the direct approach in order to be rejected);

² One can remark that in this way of calculus, the roles of inputs and outputs are reversed: specified outputs become the inputs of the calculus while the real inputs are the variables to determine.

³ For the sake of conciseness and clarity, this step has been simplified. More rigorously, another structural analysis must be conducted on the new model including the actuator model to check, in turn, its structural invertibility.

- or decide to manufacture a made-to-measure actuator if none of the off-the-shelf actuators is suitable;
- or slightly modify the specifications if the financial constraints of the project do not allow special manufactures.

In the validation phase, two cases can also happen: either the actuator chosen in step 3 suits the inputs criteria and the actuator is then validated, or the variables needed in input to fulfill the specifications do not correspond to the actuator use restrictions and the engineer must go back to the selection step. As it will be illustrated in section 3.1, in the first case, the engineer can directly conclude that the actuator is relevant for the desired behavior (and this after only two inverse calculations) and can also evaluate the possible oversizing of the actuator. On the contrary, in the second case, the engineer must even choose another actuator but, this time, the comparison between the required variables and the component limitations gives to him the origins of the undersizing (e.g. the actuator does not support such a high supply of power). Thus the engineer must go back to the selection step but with a significant guideline to follow so as to find a suitable actuator.

Thirdly, thanks to the structural analysis, the engineer can check if his problem is well-posed and, if needed, he can readapt, without any numerical calculus, his specifications to be sure that they can be reached by the chosen model structure. Thus the engineer is sure that his approach will succeed in finding a solution.

2.2 Advantage of an acausal description

From a rigorous point of view, a bond graph model initially represents a system in an acausal manner: the equations are oriented only once causality (or bicausality for inverse models) is assigned. Intuitively, the methodology proposed by the laboratory AMPERE can be applied not only for sizing problems but for other engineering contexts too: one only needs to work on the inverse model corresponding to the given problem.

Now, outside the bond graph context, a causal model is only a representation of a calculus sequence (i.e. a set of partially ordered assignments). It thus depends on the study objective and can only be used for this objective. On the contrary an acausal model is only the description of a system (i.e. a set of non-ordered implicit equations), totally independent from what oneself wants to calculate. In this way, the re-usability of models described in an acausal form

seems to be infinite while the one of causal models reduces itself only to what they are prescribed for.

As Fig. 3 shows, if the engineer chooses a causal approach, he is obliged to formulate one causal model for each problem. On the contrary, if he chooses the acausal approach, the same model can be used for all engineering problems as: analysis, sizing, control design, parametric synthesis, steady state research, ... (Fig. 4).

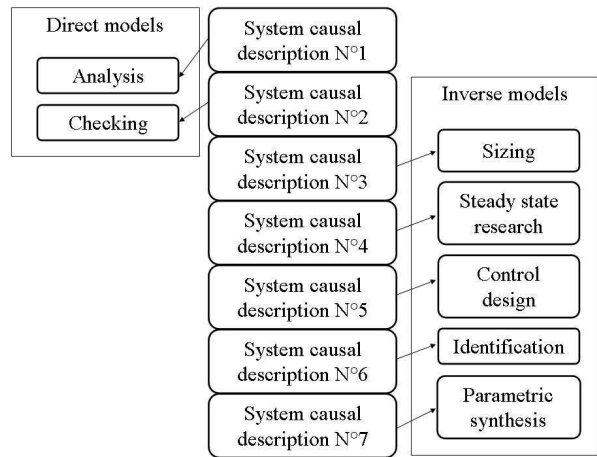


Fig. 3: System causal descriptions required for several engineering objectives

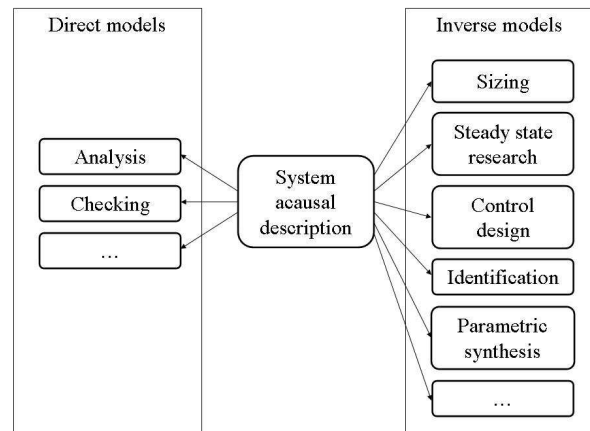


Fig. 4: Only one system acausal description required for several engineering objectives

In practice, this notion of acausality already showed its benefits especially in Modelica language and in bond graph theory.

In the Modelica context, the concept of acausality, added to the concepts of encapsulation and inheritance, enables modelling efforts to be mutualized and libraries to be obtained, libraries that are less redundant (since there is no more need to model the same component in different contexts).

In addition to this, some researches were carried out on how translating different engineering problems

into bond graph language. In fact one can remark that each of the proposed procedures starts from the acausal bond graph model so as to construct the direct (respectively inverse) model corresponding to the given problem. To quote just some of them, some works have been done on sizing problem [7][11][12], steady state research [14], parametric synthesis [15], control design [16], characterization [17] and sensibility analysis [18].

2.3 Advantage of a structural analysis

As mentioned before the first step of the AMPERE's methodology involves a structural analysis of the model which the two objectives are checking if the problem is well-posed and verifying the adequacy between the specifications and the chosen model structure.

To understand how these checks can be made, some definitions are introduced and the structural analysis is explained as well as how it can be conducted.

Concerning the concepts [9]:

- a power line is defined as a path for energy transmission between two points of the system (this is an acausal concept);
- a causal path is an ordered sequence of variables connected each one to another by the equations of the system without that a variable appears more than once in the sequence;
- an input/output power line (resp. causal path) is a power line (resp. a causal path) between an input and an output of the system;
- two power lines (resp. causal paths) are said disjoint only if there is no power (resp. no variable) in common;
- when the causality of the whole model has been assigned in order to obtain the maximum number of energy storage phenomena in integral causality, the order of a causal path is defined as the difference between the number of energy storage phenomena in integral causality and the number of those in derivative causality along this causal path.

Given a sizing problem with multiple inputs to determine from multiple specified outputs, checking if the problem is well-posed, in the sense of invertibility, thus consists in finding:

- at least one set of input/output disjoint power lines;
- and, at least one set of input/output disjoint causal paths.

If the required sets exist, then it can be concluded that the model is structurally invertible (i.e. invertible assuming that the equations of the system are locally mathematically invertible): the engineer can thus be sure that his problem is, at this stage, well-posed.

Now, on the contrary, if no set exists, it proves that the model is structurally non invertible. In that case, the procedure stops here until the problem is reformulated. This can be particularly useful for architecture synthesis. In fact if the *a priori* chosen structure does not enable the specifications to be reached, one can imagine another architecture that may satisfy the design constraints. By analyzing the input/output power lines, one can then determine the place an actuator must have in order to control a specified degree of freedom.

Finally once a good structure has been chosen and the model invertibility has been proved, the adequacy, between the specifications and the structure, can be verified. To proceed with this, one needs to check if the time derivability of each specified output is at least equal to the order of the involved input/output causal path. Not only useful for checking, this can then help to write specifications.

2.4 Methodology translation into Modelica language

If previous articles have proved the feasibility of translating a bond graph model into a Modelica model [19]-[22], the key idea here is to study to what extent a bond graph-based sizing methodology can be adapted to Modelica language. If the translation of a bond graph model into a Modelica code can be done *quasi* systematically with the BondLib library [23], the reverse operation is not so easy. Although the concepts of acausality and multi-disciplinary description seem to establish a parallel between the bond graph and the Modelica language, the conversion of a Modelica description into a bond graph model reveals itself like a harder or even impossible task.

In fact if the bond graph is intrinsically bounded to the description of the system energetic structure, nothing imposes to the modeller to depict it into Modelica language. As a proof of fact, a system can be totally described by equations gathered together into the same Modelica class, without any use of Modelica 'connect'. Moreover if 'connect' classes appear in the Modelica code, they do not necessarily represent physical energy exchanges: the Modelica modeller is totally free of choosing his variables for description.

For these reasons, the study of power lines proves to be compromised in a Modelica model and Modelica language does not seem to be suitable for the structural analysis as we have defined it above. However the interesting think of this translation tentative is to highlight that to manage a structural analysis, the engineer has to furnish a minimum set of information about the system and particularly concerning how the different physical phenomena are connected the ones to the others. Besides if we come back to the definitions relative to the structural analysis (section 2.3), one can remark that they can be formulated outside the bond graph context on condition that the concepts of energy storage/dissipative phenomena, power and energy variables be well defined. Thus one can imagine designing a sort of Modelica overlay able to depict the required information of the model.

Actually this way of doing things reveals itself more relevant since the structural analysis does not require the system equations (and so equations described in the Modelica code) but only its energy skeleton. The structural analysis pertains to a step upstream of the Modelica code writing and concerns finally directly the modelling step, where the engineer sets up the system structure and formulates the corresponding problem and specifications. Modelica can then be viewed as a complementary tool to the methodology for model exchange and reusability but not as a tool made for structural analysis.

3 MS1: an example of the methodology software implementation

To illustrate the several concepts previously described and to show how the sizing methodology can be implemented into a program, this section presents the software MS1 with its functionalities [13]. Two examples processed by it will be used to this objective: the first one concerns the case of a two-link manipulator whereas the second one involves a load actuated by a DC motor.

3.1 Methodology implementation

Structural analysis

One of the MS1 particularities is its module of structural analysis. This functionality is of course only reserved for the models described into bond graph language since the aforementioned structural analysis requires a minimum information on the system structure. Once the system is modelled into a bond graph representation and once the in-

puts/outputs of the problem are declared, the software MS1 is able to:

- search all existing input/output power lines;
- search all existing input/output causal paths;
- search all existing sets of disjoint input/output causal paths;
- determine the order of each causal paths or set of causal paths.

So, instead of doing it manually, the modeller can automatically analyze the structural properties of his model. He can conclude on his problem effectiveness and check the adequacy between the results of the structural analysis and his specifications.

Selection/validation step

Another functionality of the software MS1 is the automation of the selection step. In fact the modeller can define a place-holder for an actuator in his model and, then, the 'sizing' functionality of MS1 enables a sequence of numerical resolution to be automatically conducted. In fact, during this step, MS1 searches in a component library which actuator will be suitable for the given specifications. At the end of the calculus sequence, the engineer has a summary indicating for each actuator:

- its margins compared to what is required;
- and if the component is validated or not.

To illustrate this functionality, consider the example of a two-link manipulator (Fig. 5). This system consists of a robot made from two solid arms. The first arm is attached to the ground and to the second arm by two pivot joints which are both actuated. This robot is supposed to operate in a horizontal plane and inertias of the actuators as well as the effect of the gravity are neglected.

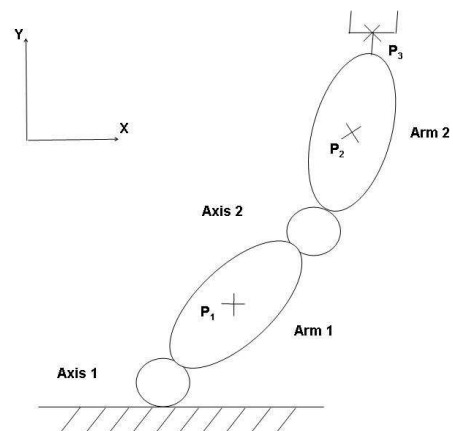


Fig. 5: Two-link manipulator system

Now consider the problem of selecting an appropriate actuating system for the axis 2 so as to the end-

effector of this robot follows a given profile in velocity⁴. The selection step, consisting of a research in the electrical drive library of the MS1 database, leads to the following two results:

- a case where the selected component does not suit the specifications (Fig. 6);

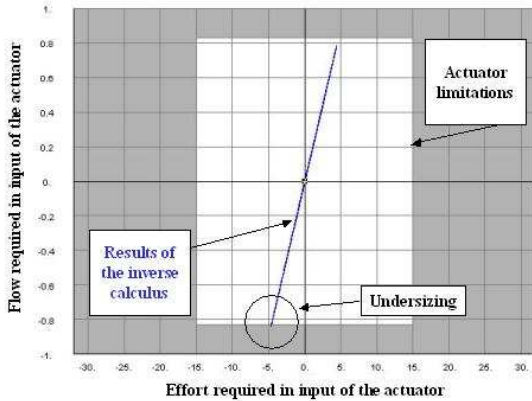


Fig. 6: Validation step: case of an undersized actuator

- a case where the selected component limitations matches with the specified trajectory (Fig. 7).

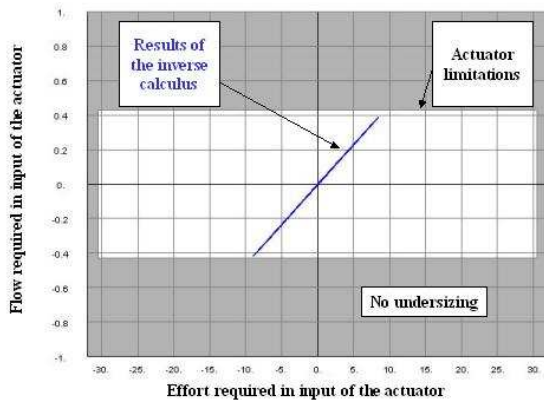


Fig. 7: Validation step: case of a suitable actuator

By representing in an (effort,flow) plane the variables required for reaching the specified output and by superimposing this curve to the manufacturer drive characteristics, one obtains a very convenient way for selecting components and for visualizing causes of under/oversizing. Moreover, as the needed variables are calculated for every instant of the dynamic specification, the engineer is able to detect at which instant the actuator overcomes its limitation and for which duration. Then he can size its component according to the dynamic criteria and, some

⁴ Even if these steps are not explicitly described here, it is assumed that the model is invertible and that the velocity profile is enough time differentiable.

manufacturer drive characteristics can be taken into account such as the ones for intermittent operation.

3.2 The Modelica code generator: illustration of the acausal description advantage

One of the advantages of the software MS1 is its concept of multi-language platform. Actually, models can be depicted into MS1 in different ways like: bloc diagram, bond graph, NMF network or algorithm. Moreover these models can also be numerically simulated by different solvers: for example, users can lead their numerical resolution by EsacpTM [24], Matlab[®] [25] or MapleTM [26]. Today one of the newest MS1 functionalities is its capability to understanding Modelica language. The software MS1 can thus:

- generate automatically Modelica code from any model described into one of the modelling languages previously quoted;
- call for the OpenModelica solver in order to proceed to the numerical resolution.

In fact the generated Modelica code is what is called 'a flat model' in the sense that it only consists of the whole equations gathered into the same class object. Thus neither heritance nor encapsulation are used here. However this model can be interpreted by any existing Modelica compiler and respects, by this way, the wish of the Modelica Association to be proprietary independent.

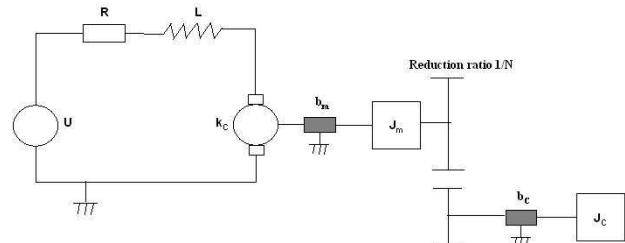


Fig. 8: DC motor actuated load system

The following example will illustrate different Modelica results generated by MS1. Consider a system consisting of a load actuated by a DC motor (Fig. 8) and suppose that the rotor shaft and the load shaft are both infinitely stiff.

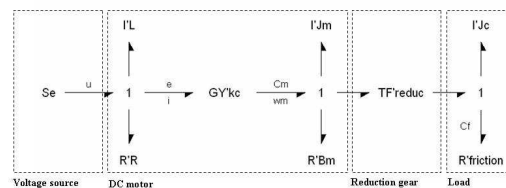


Fig. 9: Acausal bond graph model of a DC motor actuated load system

The system is modelled in terms of an acausal bond graph as shown in Fig. 9. In more details:

- the Se-element stands for the voltage source;
- the three I-elements represent the three energy storage phenomena respectively associated to the magnetic energy and the kinetic energies of the rotor and the load respectively;
- the three R-elements enable the dissipative phenomena involved respectively in the electrical circuit, on the shaft and on the load viscous type friction to be described;
- the GY-element depicts the electro-mechanical coupling;
- and the TF-element is associated to the power conserving coupling in the ideal reduction gear.

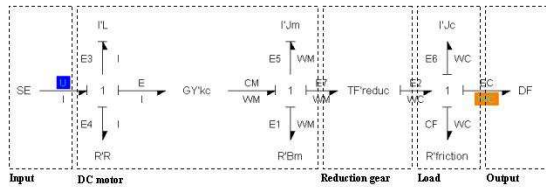


Fig. 10: Causal bond graph model of a DC motor actuated load system

Now consider a first engineering problem which the aim is to analyze the behavior of the load under a given control. Translating this problem into the bond graph language consists only just in starting from the acausal bond graph, defining the effort variable on the MSe-bond as the input, adding a Df-element representing the ideal measure of the load angular velocity and defining the corresponding flow variable as the output. This operation enables to declare which variables are known and which are to be calculated according to the given problem. The causality assignment leads to the bond graph given in Fig. 10 and the Modelica code corresponding to this problem is presented in Fig. 11.

```

class ActuatedLoad
parameter Real
  L = 0.001, R = 8.0, KC = 0.031,
  JM = 1.8E-6, N = 20.0, RC = 0.0001,
  JC = 2.E-4;
parameter Real
  G1 = 0.0, G2 = 0.0;
Real
  EC = 0.0;
Real
  U, I, E4, CM, WM, E1, WC, CF, P3,
  E2, E7, E5, E, E3, E6;

```

```

Real
  P1(start = G1), P2(start = G2);
equation
  U = 10*sin(5*time+0.0); ← INPUT
  E6 = der(P3);
  I = P1/L; E4 = I*R; CM = I*KC;
  WM = P2/JM; E1 = WM*1.0;
  WC = WM*(1/N); CF = WC*RC;
  P3 = WC*JC; E2 = EC+CF+E6;
  E7 = E2*(1/N); E5 = CM-(E1+E7);
  E = WM*KC; E3 = U-(E4+E);
  der(P1) = E3; der(P2) = E5;
end ActuatedLoad;

```

Fig. 11: Modelica code associated to an analysis problem for the DC motor actuated load system

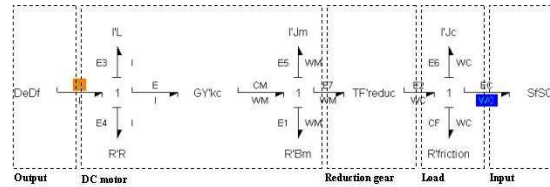


Fig. 12: Bicausal bond graph model of a DC motor actuated load system for open loop control determination

Finally consider a sizing problem where the question is to determine the open loop control of the voltage source so that the load follows a given trajectory. This time the bond graph model corresponding to this inverse problem consists in replacing the MSe-element (resp. the Df-element) by a double detector element (resp. double source element) since the roles of inputs/outputs are here reversed. Assigning bicausality results in the Fig. 12 bond graph model. The corresponding Modelica code is shown in Fig. 13.

```

class ActuatedLoad
parameter Real
  N = 20.0, JM = 1.8E-6, KC = 0.031,
  RC = 0.0001, JC = 2.E-4, L = 0.001,
  R = 8.0;
Real
  EC = 0.0;
Real
  WC, WM, E1, P2, E, CF, P3, E2, E7, CM,
  I, P1, E4, U, E5, E6, E3;
equation
  WC = 0.00193681*sin(5*time); ← INPUT
  WM = WC/(1/N); E1 = WM*1.0;
  P2 = WM*JM; E = WM*KC;
  CF = WC*RC; P3 = WC*JC;

```

```

E5 = der(P2); E6 = der(P3);
E3 = der(P1); E2 = EC+CF+E6;
E7 = E2*(1/N); CM = E1+E5+E7;
I = CM/KC; P1 = I*L;
E4 = I*R; U = E3+E4+E;
end ActuatedLoad;

```

Fig. 13: Modelica code associated to a problem of an open loop control determination for the DC motor actuated load system

One can then observe that both Modelica codes differ only by the equations concerning the input variables (respectively U for the analysis problem and WC for the sizing problem). When the Modelica ‘connect’ class will be implemented in MS1, we will obtain a model split into four classes (respectively for the dc motor, the load, the input and the output) and only those classes relative to the input and output will change between both problems.

4 Conclusion

Compared to the classical design approach, the sizing methodology, developed by the laboratory AMPERE, offers numerous benefits. In fact with the use of inverse models and structural analysis, this methodology enables the engineer to check if his problem is well-posed and to verify the adequacy of the specifications with his model structure. Moreover this tremendously decreases the number of calculus iterations since it gives, at the selection and validation steps, enough information in order to select another component in the case of an undersized one or to choose an optimal one in the case of several suitable actuators by comparing the margins of sizing.

By emphasizing the roles of acausality and multi-domain description, the aim of this paper is to ask if the methodology, originally based on the bond graph tool, may be supported by another modelling language like Modelica. After having proved the importance of acausality and structural analysis in a design approach, it has been concluded that finally the concepts used in the methodology can be defined outside the bond graph context but are not well adapted to Modelica language. In fact the notion of structural analysis requires the description of the system energy structure and thus must be conducted upstream of the Modelica code.

Here the tool MS1 enabled the feasibility of the methodology software implementation to be proved. Functionalities, like the one of automation of the structural analysis or of the component selection in an actuator library, are available. Besides this a

Modelica code generator was implemented in order to convert automatically a bond graph model into a Modelica ‘flat’ model.

In the context of the RNTL-SIMPA2 project, which the aim is to develop a Modelica compiler and integrate it into Scicos and LMS Imagine.Lab AME-Sim softwares, some researches are currently under progress for designing a module of structural analysis totally independent from any modelling language. Integrated into Scicos and more focused on the GUI, it will rely on the analysis of XML files describing the model structure. Through the GUI, the engineer will thus be guided to formulate his problem in a textual manner, describe his system in terms of energy exchanges and declare which are the known variables and the unknowns of the problem. The engineer will then be able to conduct a structural analysis (and then to apply the methodology) starting from this description, and this without knowing the bond graph theory. Results of the structural analysis will be appear in a textual manner too and a Modelica code of the problem will be eventually created.

Acknowledgement

The authors gratefully acknowledge the partial support of the French National Research Agency (ANR) through the RNTL-SIMPA2/C6E2 project.

References

- [1] <http://www.modelica.org/>
- [2] <http://www.dynasim.com/index.htm>
- [3] <http://www.lmsintl.com/>
- [4] <http://www.scilab.org/>
- [5] Ngwompo R.F., Scavarda S., Thomasset D. Physical model-based inversion in control systems design using bond graph representation, Part 1: theory, Proceedings of ImechE Journal of Systems and Control Engineering, 2001, vol. 215, pp. 95-103.
- [6] Ngwompo R.F., Scarvada S., Thomasset D. Physical model-based inversion in control systems design using bond graph representation, Part 2: applications, Proceedings of ImechE Journal of Systems and Control Engineering, 2001, vol. 215, pp. 105-112.

- [7] Ngwompo R.F., Scavarda S., Thomasset D. Bond graph methodology for the design of an actuating system: application to a two-link manipulator, IEEE International Conference on Simulation, Man and Cybernetics, Orlando, USA, 1997, pp. 2478-2483.
- [8] Ngwompo R.F. Contribution au dimensionnement des systèmes sur des critères dynamiques et énergétiques – approche par Bond Graph. Lyon, France: PhD thesis, INSA-Lyon, 1997.
- [9] Ngwompo R.F., Bideaux E., Scavarda S. On the role of power lines and causal paths in bond graph-based model inversion, in proceedings of the International Conference on Bond Graph Modeling and Simulation, New Orleans, USA, 2005, pp. 78-85.
- [10] Ngwompo R.F., Scavarda S. Dimensioning problems in system design using bicausal bond graphs, Simulation Practice and Theory, 1999, vol. 7, pp. 577-587.
- [11] Laffite J., Bideaux E., Scavarda S., Guillemard F., Ebalard M., Moutou C. Modeling in order to size an automotive power train, in proceedings of the 13th European Simulation Symposium, Marseille, France, 2001, pp. 806-812.
- [12] Mechin O., Marquis-Favre W., Scavarda S., Ferbach P. A dynamic sizing methodology in the context of an automotive application, in proceedings of the ASME International Mechanical Engineering Congress & Exposition, New Orleans, USA, 2002.
- [13] <http://www.lorsim.be/Default.htm>
- [14] Bideaux E., Marquis-Favre W., Scavarda S. Equilibrium set investigation using bicausality, Mathematical and Computer Modeling at Dynamical Systems Special Issue on Bond Graph Modeling, Taylor & Francis, 2006, vol. 12, pp.127-140.
- [15] Ngwompo R.F. Développement d'une méthodologie de dimensionnement des éléments d'un système : application au cas d'une suspension hydropneumatique. Lyon, France: Technical report, INSA-Lyon, 1999.
- [16] Bideaux E., Smaoui M., Brun X., Thomasset D. Design of a compliant positioning control using an inverse method, in proceedings of the Power Transmission and Motion Control Conference, Bath, UK, 2003, pp. 147-162.
- [17] De Giorgi R., Sesmat S., Bideaux E. Using inverse models for determining orifices mass flow rate characteristics, Sixth Japan Symposium on Fluid Power, Tskuba, Japan, 2005.
- [18] Borutzky W., Granda J. Determining sensitivities from an incremental true bond graph, in proceedings of the International Conference on Bond Graph Modeling and Simulation, Phoenix, USA, 2001, pp. 3-8.
- [19] Borutzky W., Barnard B., Thoma J.U. Describing bond graph models of hydraulic components in Modelica, Mathematics and Computer in Simulation, 2000, vol. 53, pp. 381-387.
- [20] Borutzky W. Bond graph modeling from an object oriented modeling point of view, Simulation Practice and Theory, 1999, vol. 7, pp. 439-461.
- [21] Broenink J.F. Object-oriented modeling with bond graphs and Modelica, in proceedings of the International Conference on Bond Graph Modeling and Simulation, San Francisco, USA, 1999, vol. 31, pp. 163-168.
- [22] Broenink J.F. Bond-graph modeling in Modelica, European Simulation Symposium, Passau, Germany, 1997.
- [23] Cellier F.E., Nebot A. The Modelica Bond-Graph Library, in proceedings of the 4th International Modelica Conference, Hamburg, Germany, 2005, pp. 57-65.
- [24] <http://www.ecs.dtu.dk/esacap.htm>
- [25] <http://www.mathworks.com/>
- [26] <http://www.maplesoft.com/>

Integrating Models and Simulations of Continuous Dynamics into SysML

Thomas Johnson¹Christiaan J.J. Paredis¹Roger Burkhart²

¹Systems Realization Laboratory
The G. W. Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
tjohnson6@gatech.edu chris.paredis@me.gatech.edu

²Deere & Company World Headquarters
Moline, Illinois 61265
BurkhartRogerM@johndeere.com

Abstract

In this paper, we combine modeling constructs from SysML and Modelica to improve the support for Model-Based Systems Engineering (MBSE). The Object Management Group has recently developed the Systems Modeling Language (OMG SysML™). This visual modeling language provides a comprehensive set of diagrams and constructs for modeling many common aspects of systems engineering problems, such as system requirements, structures, functions, and behaviors. Complementing these SysML constructs, the Modelica language has emerged as a standard for modeling the continuous dynamics of systems in terms of hybrid discrete-event and differential algebraic equation systems. In this paper, the synergy between SysML and Modelica is explored at three different levels: the definition of continuous dynamics models in SysML; the use of a triple graph grammar to maintain a bi-directional mapping between these SysML constructs and the corresponding Modelica models; and the integration of simulation experiments with other SysML constructs to support MBSE. Throughout the paper, an example of a car suspension is used to demonstrate these contributions.

Keywords: SysML; Modelica; model-based systems engineering; continuous dynamics; graph transformations

1 Introduction

1.1 Managing System Complexity with SysML

Contemporary systems engineering projects are becoming increasingly complex as they are handled

by geographically distributed design teams, constrained by the objectives of multiple stakeholders, and inundated by large quantities of design information. Accordingly, problems encountered during the system development process generally have more to do with the organization and management of complexity than with the direct technological concerns that affect individual subsystems and specific physical science areas [1]. If engineers cannot efficiently manage project complexity, they might overlook important design details and dependencies. Such mistakes can compromise stakeholder objectives and lead to costly design iterations or system failures.

According to the principles of model-based systems engineering (MBSE) [2], engineers can overcome these problems by replacing document-centric design methods with model-based approaches for representing and investigating their knowledge during system decomposition and definition. Models can be used to represent formally all aspects of a systems engineering problem, including the structure, function, and behavior of a system [3]. Additionally, experiments can be performed on models to eliminate poor design alternatives and to ensure that a preferred alternative meets the stakeholders' objectives. Models also facilitate collaboration by providing a common, unambiguous protocol for communicating design information.

To support MBSE, the Object Management Group has recently developed the Systems Modeling Language (OMG SysML™). SysML is a general-purpose systems modeling language that enables systems engineers to create and manage models of engineered systems using well-defined, visual constructs [4]. Instead of developing SysML as an original design, the OMG adapted the successful Unified Modeling Language (UML) to the systems engineering

field. UML is most commonly used during the development of large-scale, complex software for various domains and implementation platforms [5]. To support an application base that extends beyond software engineering, SysML reuses and extends a subset of UML 2.1 constructs:

- it extends UML classes into *blocks*;
- it enables *requirements modeling*;
- it supports *parametric modeling*;
- it extends UML dependencies into *allocations*;
- it reuses and modifies UML *activities*;
- it extends UML standard ports into *flow ports*.

Through these extensions, SysML is capable of representing many common, yet essential aspects of both system hardware and software.

1.2 Modeling System Behavior with SysML

The knowledge captured in a SysML model is intended to support the specification, analysis, design, and verification and validation of any engineered system [4]. As a result, SysML is commonly used to model system requirements, tests, structures, functions, behaviors, and their interrelationships. Although all of these models are important for ensuring project success, behavioral models are arguably the most important. If the system does not behave in a way that satisfies stakeholder objectives, then it is useless regardless of its other aspects.

SysML currently depicts system behavior using the following language constructs:

- *Activity diagrams* describe the inputs, outputs, sequences, and conditions for coordinating various system behaviors;
- *Sequence diagrams* describe the flow of control between actors and a system or its components;
- *State machine diagrams* are used for modeling discrete behavior through finite state transition systems;
- *Parametric diagrams* allow users to represent mathematical constraints amongst system properties.

The first three of these modeling constructs promote causal behavioral modeling in terms of discrete events. The last one enables a user to model equations (called “constraints” in SysML) that establish mathematical relationships between system properties. In this paper, the focus is on *parametric diagrams* and specifically on the representation of the continuous dynamics of engineered systems within parametric diagrams. Such models are composed of

differential algebraic equation (DAE) systems that represent the exchange of energy, signals, or other continuous interactions between system components. By relying on Modelica syntax and semantics, we demonstrate how such DAE systems can be modeled with only a few extensions to the basic SysML constructs (see Section 4). SysML then serves as an integration framework in which detailed Modelica models can be related to other types of systems engineering knowledge (see Section 6). The integration between SysML and Modelica creates a significant synergy: SysML benefits from the detailed Modelica semantics for representing DAE systems combined with discrete events; Modelica benefits from the broader information modeling context provided in SysML, a context that is crucial for establishing formal, unambiguous communications between systems engineers, disciplinary designers and systems analysts. To maintain consistency between the Modelica models and their corresponding abstractions in SysML, we introduce the use of triple graph grammars (TGGs) [6] to specify transformations between the two forms of models (see Section 5).

2 Related Work

The need to describe system behavior in terms of equations or constraints has been previously recognized in the work on Constrained Objects (COB’s) [7, 8]. COBs provide both a graphical and lexical representation of algebraic relationships that can be used to tie design models to analysis models in a parametric fashion. These COBs recently served as the basis for the development of the SysML parametric diagrams [4]. By establishing a mapping between COBs and SysML, the integration and execution of engineering analyses (such as structural finite element analyses) within the context of SysML has been demonstrated [9]. This paper extends this past work on COBs by focusing on the modeling and simulation of the continuous dynamics of systems as defined in Modelica models.

Recently, Fritzson and Pop [10] have worked on the integration of UML/SysML and Modelica to provide support for modeling and simulating continuous dynamics. They have created a UML profile called ModelicaML that enables users to depict a Modelica simulation model graphically alongside UML/SysML information models. The ModelicaML profile reuses several UML and SysML constructs, but also introduces completely new language constructs. Such constructs are the Modelica class diagram, the equation diagram, and the simulation diagram.

Nytsch-Geusen [11] developed a specialized version of UML called UML^H. This version is used in the graphical description and model-based development of hybrid systems in Modelica. The author presents hybrid system models as Modelica models that are based on DAEs combined with discrete state transitions modeled with the Modelica statechart extension. Using a UML^H editor and a Modelica tool that supports code generation, Modelica stubs can be automatically generated from UML^H diagrams so that the user must only insert the equation-based behavior of the system in question. In this paper, the capabilities of ModelicaML and UML^H are further extended by demonstrating the integration of continuous dynamics models with other SysML constructs for requirements, structure, and design objectives, and by demonstrating the translation between SysML and Modelica through the use of TGGs.

3 An Introduction to SysML: The Car Suspension Model

Before discussing the approach for modeling continuous dynamics and simulations in SysML, this section reviews some important SysML constructs and introduces the example problem used throughout this paper.

3.1 SysML Blocks

The primary modeling unit in SysML is the *block*. As described in chapter 8 of the SysML specification [4], a block is a modular unit of a system description. A block can represent anything, whether tangible or intangible, that describes a system, process, function, or context. When combined together, blocks define a collection of features that describe a system or other object of interest. Hence, blocks provide a means for an engineer to decompose a system into a collection of interrelated objects.

All block declarations occur in a *Block Definition Diagram* (BDD). A BDD is used to define block features and the relationships between blocks or other SysML constructs. Figure 1 depicts the definition of a car and its suspension. A car is obviously composed of more subsystems and components, but Figure 1 is sufficient for the sake of demonstration. SysML allows a modeler to omit elements of the underlying information model that detract from the main intent of a diagram.

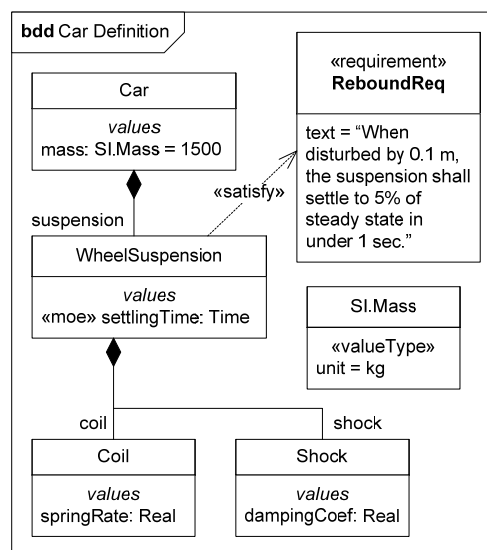


Figure 1. The SysML car suspension model.

3.2 SysML Properties

A SysML property describes a part or characteristic of a block and consists of a named value of a specified type. In Figure 1, two important categories of properties are depicted. The first kind of property is a *part property*. Part properties represent a subsystem or component of a system and must be typed by a block. Part properties can be depicted in the *parts compartment* of a block or using a *composition association*. A composition association is depicted using a black diamond with a tail. The property name appears at the tail end of the association. For example, the block *Car* in Figure 1 owns a part property named *suspension* of type *WheelSuspension*.

The second kind of property is a *value property*. A value property appears in a block's *values compartment* and represents a quantifiable characteristic of a block (e.g. mass, length, velocity) and must be typed to a SysML *value type*. A value type is a special modeling element (similar to a block) used to assign the units of measure and dimension declared in its definition. For example, *Car* in Figure 1 has a value property *mass* which is typed to the value type *SI.Mass* to supply units of kilograms.

3.3 UML Stereotypes

A *stereotype* is a UML construct used to create customized classifications of modeling elements. Stereotypes are defined by keywords that appear inside of guillemets. These customization constructs extend the standard elements to identify more specialized cases important to specific classes of appli-

cations. Most SysML constructs have been defined as UML stereotypes, and users are allowed to create additional stereotypes to capture the specialized semantics of a particular application domain. An example of a stereotype is illustrated in Figure 1. The stereotype «*moe*» applied to the *WheelSuspension*'s value property *settlingTime* indicates that it is a measure of effectiveness.

3.4 SysML Requirements

A SysML *requirement* is used to represent a textual requirement or objective for a system, subsystem, or component. Requirements are shown with the «*requirement*» stereotype and optionally have a compartment for displaying text and identification fields. Requirements are related to other modeling elements using various dependencies such as the *satisfy* and *verify* dependencies.

4 Modeling Continuous Dynamics in SysML

In this section, the approach to modeling continuous dynamics in SysML is presented. The approach builds on the initial modeling foundation outlined in [12]. Rather than elaborating upon every detail, only the most important modeling constructs are discussed.

4.1 Objectives

A model is valuable if it increases a decision maker's ability to design a better system at an acceptable cost [13]. As explained later in this section, the continuous dynamics modeling constructs will provide value if they meet the following objectives:

- Enable the integration of continuous dynamics models into broader SysML models;
- Facilitate the execution (i.e., simulation) of these continuous dynamics models;
- Encourage model reuse;
- Facilitate efficient stakeholder communication.

The intent of these objectives is to strike an appropriate balance between the benefits expected from developing a model and the costs of encoding the required information.

Model integration is essential for managing system complexity through recognition and establishment of dependencies and associations between models of continuous dynamic system behavior and other models of system behavior, structure, or func-

tionality. SysML is a language for describing systems engineering information and knowledge, but is by itself not executable—model execution is relegated to an editing and execution tool. To be effective, it is therefore important to establish seamless connections between SysML and simulation tools. Model reuse is another imperative for realizing significant reductions in project resource expenditures. Finally, using a unified approach for representing continuous dynamics in SysML establishes a protocol for unambiguous communication of behavioral information between designers operating in various engineering disciplines.

4.2 Modelica as a Foundation

When creating a formal approach for representing continuous dynamics in SysML, Modelica provides a strong foundation. Modelica has emerged as the language of choice for expressing continuous dynamic system behavior. It is better structured and more expressive than most alternatives such as VHDL-AMS [14] or ACSL [15]. In addition, both SysML and Modelica are similar in that they use base modeling elements that adhere to the principles of object-oriented modeling. Both languages also encourage model reuse through acausal equation-based modeling. Unfortunately, enough differences exist such that a direct one-to-one mapping is not possible. Since SysML is intended to be a general modeling language, some of the specialized semantics of Modelica do not have a direct equivalent in SysML. To overcome these differences, our approach has been to find a good balance between converting some implicit Modelica semantics into explicit constraints in SysML or, when that is not possible, extending the SysML constructs through stereotypes.

4.3 Model Declaration

When modeling continuous dynamic system behavior, a modeler must first declare the model that represents the system of interest. This involves specifying the blocks and properties needed to decompose the system to an appropriate level of abstraction. The level of abstraction is determined by the amount of detail needed to perform an acceptable system analysis. This declaration approach is analogous to creating Modelica classes that own components and variables typed to other class definitions.

To illustrate model declaration, Figure 2 displays the declaration of a continuous dynamics model of a Mass-Spring-Damper (MSD) system. This model will be used in Section 6 to perform a behavioral

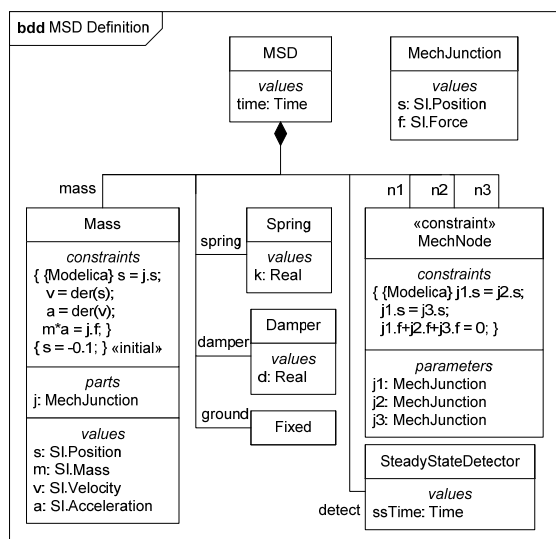


Figure 2. BDD of the *MSD* continuous dynamic system behavior model.

analysis on the car suspension model from Figure 1. The *MSD* system is composed of a mass, spring, damper, fixed position (i.e. ground fixture), and a detector that determines system settling time. The block *MSD* represents the declaration of the *MSD* system while the other blocks (*Mass*, *Spring*, *Damper*, *SteadyStateDetector*, *Fixed*, and *MechJunction*) represent the definitions of the system components.

Upon declaring the necessary models, their properties must be identified. Figure 2 depicts the declaration of both the part and value properties. *MSD* is attributed with the *mass*, *spring*, *damper*, *ground*, and *detect* part properties typed to the *Mass*, *Spring*, *Damper*, *Fixed*, and *SteadyStateDetector* block definitions, respectively. While *MSD* has no value properties, most of the block definitions to which its part properties are typed contain value properties. For example, *Mass* contains a value property *m* typed to the value type *SI.Mass*.

4.4 Model Interface

To interact with other models, a given model must have a well-defined interface. Models used in the description of a system's continuous dynamic behavior generally interact using exposed *across* and *through* variables [16]. Since across and through variables are the only means of interaction, they should be encapsulated inside of reusable blocks that are typed to the part properties of another block. These part properties are then exposed to other system components and subsystems. This type of interface is similar to the usage of Modelica connectors.

To illustrate the declaration of a model interface, Figure 2 depicts a block named *MechJunction*. This is a reusable block that encapsulates position and force value properties corresponding to translational across and through variables. To define the interfaces for each component of *MSD*, the appropriate number of part properties are declared for each component and then typed to *MechJunction*. For example, *Mass* has one part property *j* typed to *MechJunction*.

4.5 DAE-Based Internal Behavior

To define a model's DAE-based internal behavior, Modelica relies on equations declared in the equation clause of a given class. Similarly, this is accomplished by placing SysML *constraints* on a given block. A constraint is simply the representation of an equation that constrains a block's value properties. Constraints appear between braces and are displayed in a block's constraints compartment. To model initial conditions, a constraint can be assigned the *«initial»* stereotype. This stereotype is an extension to SysML; it can only be assigned to constraints and implies that the constraint only holds true at the beginning of a simulation.

Usages of constraints and the *«initial»* stereotype are shown in Figure 2. The internal behavior of the block *Mass* is defined using four regular constraints and one initial constraint. Note that the constraints explicitly refer to the Modelica language, but other syntax could be used according to the modeler's preferred executable language.

4.6 Energy and Signal Flow between System Components

To model the flow of energy through a system and its components, a means of interaction must be provided to the interface part properties described in Section 4.3. Generally, the flow of energy in a system is described using the equivalent of Kirchhoff's circuit laws: at a connection, all across variables are equal, while all the through variables add up to zero. While this is modeled implicitly in Modelica using *connect clauses*, our SysML modeling approach explicitly models the interaction with reusable *constraint blocks*. As defined in the SysML specification [4], a constraint block is a specialized form of the SysML block and is intended to package commonly used equations in a reusable, parameterized fashion. Constraint blocks can be identified by the *«constraint»* stereotype that appears in their namespace compartment. To use the definition of a constraint block, another block or constraint block can

declare a *constraint property* and assign the type to a constraint block. Using a SysML *parametric diagram*, the parameters used in the definition of the constraint can be bound to the properties of another block or constraint block using *binding connectors*. A binding connector implies a *pure equality* constraint between two objects. If the objects are part properties, then all of the sub-properties belonging to each part are equal. It is this difference between the semantics of SysML binding connectors and Modelica connections that necessitates the inclusion of an explicit node constraint block in SysML.

Figure 2 shows the definition of a constraint block named *MechNode*. This constraint block has three parameters *j1*, *j2*, and *j3* of type *MechJunction*. The across and through variables of these parameters are subject to the three packaged constraints that describe Kirchhoff's circuit laws for a translational mechanical system. MSD owns three constraint properties typed to *MechNode* to enable the interaction of its part properties. Figure 3 displays a parametric diagram that depicts the part interactions as a result of binding usages of *MechJunction*.

5 SysML and Modelica Integration

Currently, system engineering problems are solved using a wide range of domain-specific modeling languages. Moreover, it is unlikely that a single unified modeling language will be able to model in sufficient detail the large number of system aspects addressed by current domain-specific languages. One should not “reinvent the wheel” by creating an all-encompassing systems engineering language capable of modeling and simulating every aspect of a system. On the other hand, managing a large number of models in different languages also poses problems, including communication ambiguity and the preservation of information consistency. To alleviate these problems, a model integration framework is needed for managing the various modeling languages used to solve systems engineering problems.

SysML can provide an answer to this need for model integration. Using SysML, a modeler can abstract a domain-specific language to a level that permits its interaction with other system models. For example, a Modelica model is an excellent way to capture hybrid discrete/DAE-based system behavior, but is not capable of modeling system structure or requirements. Using the modeling approach outlined in Section 4, a modeler can abstract a Modelica

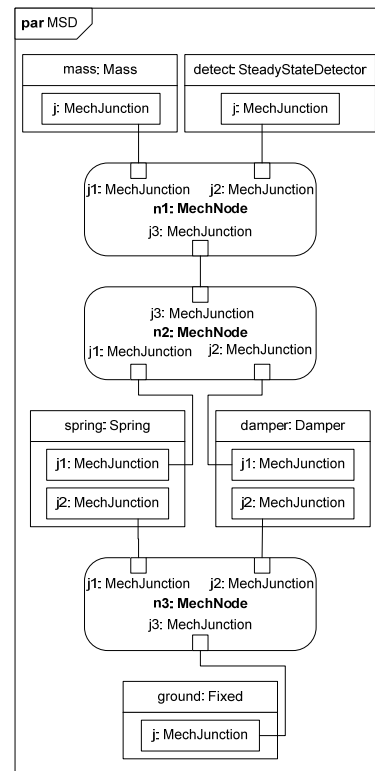


Figure 3. Parametric diagram of the *MSD* model.

model into SysML syntax to represent dependencies and associations with other system models¹.

While SysML is a valuable integration tool, much of that value could be detracted if engineers must manually transform domain-specific models into SysML and vice-versa. In the case of continuous dynamics models, we need an approach for accomplishing automated, bidirectional transformations between the SysML and Modelica languages.

Many methods exist for completing model transformations between two or more modeling languages (metamodels). Two common transformation tools are OMG's Queries/Views/Transformations (QVT) [17] and TGGs [6].

The QVT specification provides a set of languages for querying a source model that complies with a source metamodel and transforming it into a target model that complies with a target metamodel. Two QVT languages, *Relations* and *Core*, are used

¹ Dependencies and associations are UML constructs for expressing types of relationships between information objects.

to declaratively model the relationships between source and target metamodels at different levels of fidelity. The *Operational Mappings* language is then used to perform imperative transformations based on the relationships depicted in the *Core* or *Relations* languages. Overall, QVT is a powerful and widely accepted model transformation tool; however, the imperative nature of the *Operational Mappings* language hampers bidirectional transformations.

TGGs are similar to QVT in intent but are declarative by nature. Accordingly, TGGs are particularly useful for completing complex, bidirectional model transformations. In a TGG, the metamodels for the source and target languages are defined as graphs. The mapping between the two languages is then represented as a set of graph transformation rules applied to a third graph: a *correspondence graph*. For example, a SysML block would be related to a Modelica class using a correspondence entity named *block2class* with one relation pointing to the *block* entity (in the SysML metamodel graph) and one to the *class* entity (in the Modelica metamodel graph). By querying a model space containing SysML or Modelica models, transformations are performed until the model space complies with the specified TGG.

Due to the declarative, bidirectional nature of TGGs, one set of graph transformation rules can be used to transform SysML models into Modelica and vice-versa. Although a TGG is used for this transformation, others have shown that QVT is equally expressive and capable [18]. The TGG and graph transformation rules have been encoded in the Visual Automated Model Transformations (VIATRA) [19] framework. VIATRA enables modelers to create models in a declarative fashion and use pattern recognition to complete graph transformations in a sequential fashion using machines. To demonstrate this TGG, a Java plug-in for Eclipse has been implemented to transform SysML models developed in the Embedded Plus (E+) modeling environment into Modelica models using the OpenModelica [20] compiler (OMC) and Modelica Development Tooling (MDT) plug-in for Eclipse. The functionality of this plug-in is depicted in Figure 4.

6 Modeling Simulations in SysML

In the context of model-based systems engineering, models and simulations allow systems engineers to investigate and predict the behavior of system alternatives without the need for physical prototyping. For example, a continuous dynamics model of a

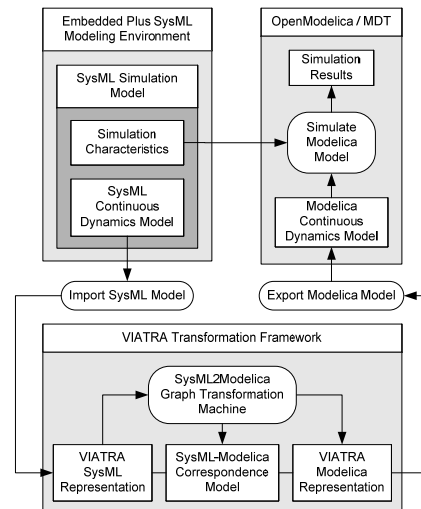


Figure 4. Functionality of the SysML-to-Modelica transformation Eclipse plug-in.

MSD can be used to simulate and predict the behavior of a car suspension alternative. This section describes how a continuous dynamics model can be related to other relevant design information in SysML: binding of model parameters in a *model context*; defining an experiment performed on a model in a *simulation*; defining a measure of effectiveness as the result of a simulation; and using an *abstracted simulation* in the context of design optimization.

6.1 Defining the Model Context

In systems engineering, a continuous dynamics model is always used in a particular model context. Within this model context the elements of the system structure are bound to the corresponding elements of the analysis model. In current practice, engineers do not always distinguish between the physical structure or system topology and the corresponding system behavior. For instance, it is common practice to use an electric circuit diagram as the representation for defining both the circuit topology as well as the behavior of the circuit in a SPICE simulation. As systems become more complex there often is a need to represent a system by multiple simulation models, corresponding to different levels of abstraction or different disciplinary perspectives. The use of an explicit model context as suggested here facilitates the preservation of consistency amongst all the separate models.

To relate the structure to the behavior, a *model context* block is defined with two part properties: one usage of the system model and one usage of the analysis model. If mathematical relationships be-

yond simple equivalence exist between the known elements of the system model and the corresponding elements of the analysis model, additional constraint blocks can also be defined. Finally, a parametric diagram of the model context block is created to bind the known system elements to the corresponding analysis elements.

In the lower portion of Figure 5, the block *ModelContext* is defined as owning usages of *MSD*, *Car*, and a constraint block named *MassRelation*. In Figure 6, a corresponding parametric diagram is shown establishing a relationship between the MSD and car masses. Inside of this parametric diagram, *msd.mass.m* is defined as one quarter of the mass of *mcCar.mass* by connecting them to the appropriate parameters on the constraint property *massRel*.

6.2 Modeling the Simulation

A simulation is an experiment performed on a computational model [21]. Before a simulation can be performed, the experiment needs to be completely defined: the initial values and boundary values, the outputs to be observed, and potentially the process steps one should go through in the experiment (e.g., time traces of external inputs). From a modeling perspective, all of these aspects can be captured in the model itself or in extensions of the model defined using the same Modelica/SysML constructs described in Section 4. One can therefore assume that the “model” as defined in the model context is fully specified — all the parameters are bound to values and the set of system equations is non-singular. Under those assumptions, the only additional information that needs to be provided is the start and end

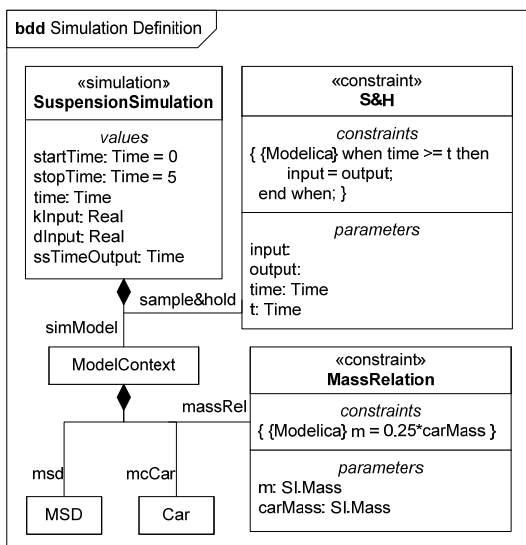


Figure 5. BDD of the *SuspensionSimulation* block.

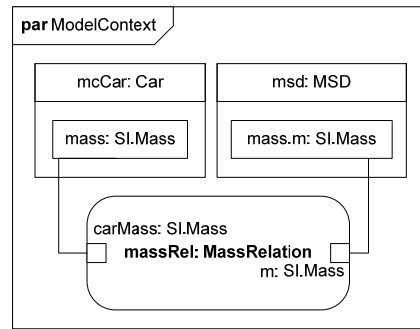


Figure 6. Parametric diagram of the *ModelContext*.

time of the simulation.

To make the semantics of a simulation explicit in SysML, we have defined a «simulation» stereotype. As is illustrated in Figure 5, this stereotype requires the inclusion of a *time* property, which represents the simulation time; *startTime* and *stopTime* properties; and a *simModel* block. The meaning of the stereotype is then that all the properties in the *simModel* are evaluated as a function of *time* from *startTime* to *stopTime*. Note that this stereotype completely defines a simulation experiment in a fashion that is independent of any particular simulation solver. In addition, note that Modelica semantics differ from SysML semantics which require the explicit definition of a local simulation time property to which all time-varying system properties can be bound.

6.3 Abstracting the Simulation

A simulation as defined in the previous section allows a systems engineer to define an experiment in which the system behavior can be observed. However in systems engineering, simulations are often used to make decisions. In that case, the same experiment is often performed on multiple variations of the same system — the design or decision alternatives. It then becomes important to abstract this simulation formally by clearly defining the inputs (the properties that can take on different values from one simulation run to the next), and the outputs (the properties that are of interest to the design, for instance, a measure of effectiveness that drives a design optimization). The relationship between inputs and outputs of the simulation can then itself be considered as a model. Unlike the model of the system, this input-output model is an algebraic relationship, albeit a very complex one that requires running the entire simulation to compute the outputs from the inputs. When abstracting (or “wrapping”) a simulation in this fashion in support of decision making, it is justifiable to assume that the outputs of the simulation are scalar quantities (decisions can only be made

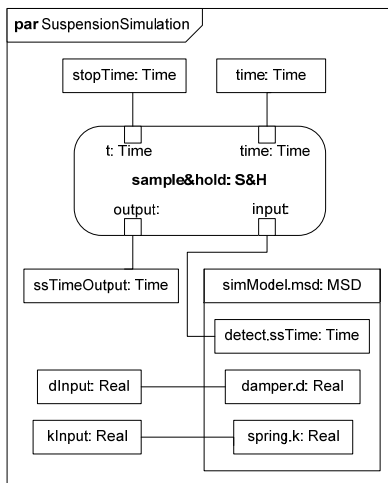


Figure 7. Parametric diagram of *SuspensionSimulation*.

based on scalars because vectors cannot be rank-ordered [22]). Sometimes this requires that one include additional modeling elements in the continuous dynamics model to define these scalar measures of effectiveness. For instance, in the BDD in Figure 5 and the corresponding parametric diagram in Figure 7, the suspension simulation has been abstracted into an input-output model with inputs as the decision variables, *dInput* and *kInput* (bound to the damping and stiffness of the suspension), and an output as the measure of effectiveness, *ssTimeOutput* (the steady-state time of the mass-spring-damper system). The output has been bound to a model property through a sample and hold constraint property, *sample&hold*, making explicit that the output takes on the value of the time-varying property *detect.ssTime* when the simulation time equals *stopTime*. In general, more complex models may be necessary to relate scalar outputs to time-varying simulation properties.

6.4 Embedding a Simulation into an Analysis

Once a simulation has been abstracted into an input-output model, it can be used in support of analyzing system alternatives with respect to stakeholder requirements and measures of effectiveness, as is illustrated in Figures 8 and 9. Analyses generally verify that a system alternative meets a certain system requirement, which can be modeled explicitly using the «verify» dependency. A parametric diagram of that block can be used to connect the system alternative to the simulation, as is illustrated in Figure 9. Instead of binding the simulation inputs and outputs directly to the corresponding value properties of the system alternative, one could also define an optimization problem in which the stiffness and damping are optimized with respect to one or more

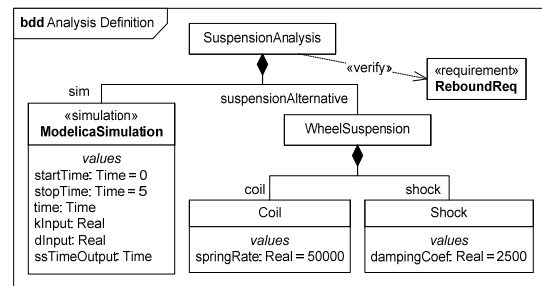


Figure 8. BDD of the *SuspensionAnalysis* block.

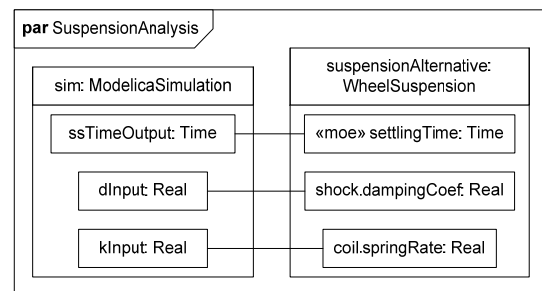


Figure 9. Parametric diagram of *SuspensionAnalysis*.

measures of effectiveness. Whenever there is a need for repeated evaluation of the simulation with different inputs, it is desirable to embed the simulation explicitly in an analysis context as is shown in Figure 8.

7 Discussion and Closure

In this paper, we have introduced an approach for combining SysML and Modelica in a synergistic fashion. No single language or formalism can possibly capture all of the knowledge and information needed to solve systems engineering problems. While Modelica is well-suited for describing the dynamic behavior of complex systems, it offers no support for relating that behavior to stakeholder requirements. Similarly, SysML allows one to define the high-level relationships between requirements and functional, physical and operational architectures of a system, but lacks the detailed semantics to capture for instance geometry. It is therefore crucial that capabilities are developed for relating in a formal framework the different knowledge representations commonly employed in systems engineering problems. SysML provides the foundation for making a first step in that direction. The general-purpose and adaptable nature of the language enables system engineers to interrelate their preferred knowledge representations. In addition, formal metalevel mappings as described by TGGs provide a promising founda-

tion for bidirectional mappings between the different knowledge representations.

Using the modeling approaches described in this paper, engineers will be more capable of managing system complexity through the modeling of dependencies between continuous dynamic system behavior and other system aspects. Additionally, the mapping of SysML to Modelica and the resulting transformation abilities enable engineers to describe their systems at a higher level of abstraction while still maintaining the benefits of executable knowledge representations.

In this paper, the intent has been to take advantage of SysML's adaptability and to make a step towards the unification of various modeling formalisms. While the continuous dynamics modeling approach described in this paper builds on the Modelica language, it still maintains a certain language independence thanks to the general, declarative nature of Modelica. TGGs could be developed to map SysML to the syntax of other languages, with the restriction that when mapping to a causal, procedural modeling language, a compiler must be used to assign causalities and sort the equations.

The ongoing efforts towards the unification of engineering knowledge representations in SysML are exciting steps for the systems engineering community. Utilizing and increasing the abilities of SysML promises to improve the current state of systems engineering and bring to fruition the benefits of MBSE.

Acknowledgements

This work has been funded by Deere & Company. Additional support was provided by the ERC for Compact and Efficient Fluid Power, supported by the National Science Foundation under Grant No. EEC-0540834. The authors would also like to thank Sanford Friedenthal, Leon McGinnis and Russell Peak for the discussions that helped crystallize the ideas presented in this paper.

References

- [1] Sage, A. P., and Armstrong Jr., J. E., 2000, *Introduction to Systems Engineering*, John Wiley & Sons, Inc., New York, NY.
- [2] Fisher, J., 1998, "Model-Based Systems Engineering: A New Paradigm," *INCOSE Insight*, 1(3)
- [3] Gero, J. S., 1990, "Design Prototypes: A Knowledge Representation Schema for Design," *AI Magazine*, 11(4), pp. 26-36.
- [4] Object Management Group, 2007, "OMG Systems Modeling Language Specification," <http://www.omg.org/cgi-bin/doc?ptc/07-09-01>.
- [5] Booch, G., Jacobson, I., and Rumbaugh, J., 2005, *The Unified Modeling Language User Guide*, Addison-Wesley Professional.
- [6] Schürr, A., 1994, "Specification of Graph Translators with Triple Graph Grammars," in *WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*.
- [7] Peak, R. S., and Wilson, M. W., 2001, "Enhancing Engineering Design and Analysis Interoperability Part 2: A High Diversity Example," *First MIT Conference Computational Fluid and Structural Mechanics (CFSM)*, Cambridge, Massachusetts, USA.
- [8] Peak, R. S., Burkhart, R. M., Friedenthal, S. A., Wilson, M. W., Bajaj, M., and Kim, I., 2007, "Simulation-Based Design Using SysML-Part1: A Parametrics Primer," in *INCOSE Intl. Symposium*, San Diego, CA.
- [9] Peak, R., Friedenthal, S., Moore, A., Burkhart, R., Waterbury, S., Bajaj, M., and Kim, I., 2005, "Experiences Using SysML Parametrics to Represent Constrained Object-Based Analysis Templates," *7th NASA-ESA Workshop on Product Data Exchange (PDE)*, Atlanta, GA, USA.
- [10] Pop, A., and Akhvediani, D., and Fritzson, P., 2007, "Towards Unified Systems Modeling with the ModelicaML UML Profile," in *International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping University Electronic Press, Berlin, Germany.
- [11] Nytsch-Geusen, C., 2007, "The Use of UML within the Modelling Process of Modelica-Models," in *International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping University Electronic Press, Berlin, Germany.
- [12] Johnson, T. A., Paredis, C. J. J., Burkhart, R., and Jobe, J. M., 2007, "Modeling Continuous System Dynamics in SysML," in *2007 ASME International Mechanical Engineering Congress and Exposition*, ASME, Seattle, WA.
- [13] Keeney, R. L., 1994, "Creativity in Decision Making with Value-Focused Thinking," *Sloan Management Review*, 35(4), pp. 33-41.

- [14] Christen, E., and Bakalar, K., 1999, "VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **40**(10), pp. 1263-1272.
- [15] Mitchell, E. E. L., and Gauthier, J. S., 1976, "Advanced Continuous Simulation Language (ACSL)," *SIMULATION*, **26**(3), pp. 72-78.
- [16] Paynter, H., 1961, *Analysis and Design of Engineering Systems*, MIT Press, Cambridge, MA.
- [17] Object Management Group, 2007, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," <http://www.omg.org/docs/ptc/07-07-07.pdf>.
- [18] Greenyer, J., Kindler, E., 2007, "Reconciling TGGs with QVT," in *Model Driven Engineering Languages and Systems, MODELS 2007*, Springer, Berlin / Heidelberg.
- [19] Varró, D., 2003, *VIATRA: Visual Automated Model Transformation*, Thesis, Department of Measurement and Information Systems, University of Technology and Economics, Budapest.
- [20] Fritzson, P., et al., , 2007, "OpenModelica System Documentation," <http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.4.3/doc/OpenModelicaSystem.pdf>.
- [21] Fritzson, P., 2004, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press, Piscataway, NJ.
- [22] Keeney, R. L., and Raiffa, H., 1976, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, Jon Wiley and Sons, New York.

Modelica library for logic control systems written in the FBD language

Alberto Leva, Filippo Donida, Marco Bonvini*, Lorenzo Ravelli*

Dipartimento di Elettronica e Informazione, Politecnico di Milano

Via Ponzio, 34/5 – 20133 Milano, Italy

{donida,leva}@elet.polimi.it

*former student at the Politecnico di Milano

Abstract

The paper describes a Modelica library for the simulation of logic control systems written in the FBD (Functional Block Diagram) language as defined in the IEC61131.3 standard. The library contains not only strictly logic blocks, but also the main types of industrial PID controllers. Models of different complexity levels are included, so that the user can specify a control system as a continuous-time model (for fast simulation to check whether or not a control strategy solves the problem at hand) or an event-based one (for precise evaluation of the algorithms' behaviour).

1. Introduction

In many control domains, particularly – but not exclusively – in the process control field, a correct representation of the control system connected to the plant being investigated is of paramount importance [7, 1, 3, 18, 19, 8, 9, 4, 11, 13, 17, 21, 15]. In many cases, the structuring and the subsequent tuning of that control system is even the main goal of the simulation activity; and also if control commissioning is not the primary purpose of the simulation, having a correct and realistic control representation is always important in order to draw meaningful conclusions.

Nowadays, more and more control systems are implemented adhering to the IEC61131.3 standard [1, 2, 6, 7, 20, 10, 11, 14, 16], that defines five programming languages (Ladder Diagram or LD,

Sequential Functional Chart or SFC, Functional Block Diagram or FBD, Structured Text or ST, Instruction List or LD) basically oriented to logic control, although most systems adhering to the standard also offer modulating control functions. In the last years, the IEC61131.3 standard has become very popular in the arena of PLC programming, therefore spreading out in a vast number of contexts and applications [2, 10, 8, 16, 15, 5].

As such, having the IEC61131.3 standard available in the Modelica environment is of great help, for at least two reasons. First, if an *industry* standard is *uniformly* adopted, there is (ideally) no room for ambiguities in the communication between the people who own and/or run the plant, and the analysts who create the simulator and realise the necessary studies (some issues may still arise owing to the fact that virtually every

standard is the result of a compromise, and therefore very frequently exists also in the form of so-called “dialects”, but addressing that problem is apparently beyond the scope of this research). Second, the solutions found at the simulation level are deployed to the target control architecture in a very straightforward way.

For the reasons above, a free (GPL) FBD Modelica library is being developed at the Politecnico di Milano. The present state of that library is described in this paper, that is organised as follows. Section 2 describes the organisation of the library, briefly list its contents, and presents some selected blocks with a minimum of detail. Section 3 discusses two examples. The first aims at showing the importance of having the regulators described both at a simplified (continuous-time) and at a detailed (event-based) level. The second shows some library blocks applied to the control of a small manufacturing system, to illustrate how the obtained Modelica schemes are easily understood by people developing code for the typical industrial control architectures. Finally, section 4 reports some conclusions, and the future plans of the research.

2 Library organisation

The library comes in a single Modelica package named *FBD*, and organised in subpackages as sketched below:

- the *FBD.OneBitOperation* subpackage implements basic logical operations,
- the *FBD.CompareOperation* subpackage implements comparisons (the $<$, $>$, $>=$, $<=$, $=$ operators) on the Integer and Real types,
- the *FBD.Counter* subpackage provides

up/down counters,

- the *FBD.MathOperation* subpackage implements the basic mathematical instructions,
- the *FBD.Timer* subpackage provides timers (and is similar to the Counter one),
- the *FBD.NBitOperation* subpackage implements logical operation on arrays of bits,
- the *FBD.LinearSystems* subpackage provides linear, time invariant dynamic systems in the continuous and discrete time, as typically specified in IEC-compliant control code development environments,
- the *FBD.IndustrialController* subpackage contains several industrial controllers, including of course several types of PID,
- the *FBD.Test* subpackage contains test simulators for each FBD block, individually, to allow for a precise comprehension of its functionalities,
- and finally the *FBD.Applications* subpackage provides some examples of use of the FBD blocks of the library.

For obvious space reasons we do not describe the blocks here, referring the interested reader to the library documentation. A couple of remarks are however worth some lines.

First, for *every* component a “test” model is provided, to allow the user to fully understand how that component works, and possibly disambiguate situations where the available specifications are not fully univocal; everyone wishing to extend the library (contributions are of course welcome in the GPL spirit) is strongly

encouraged to do the same.

Second, especially for regulators, both continuous-time and event-based models are present. The former type of model allows for faster simulation, and is the choice of election when the purpose is to check the correctness of a control *strategy*. The latter is apparently less time-efficient, but allows to check the behaviour of a control *algorithms*. The library therefore allows to perform both types of simulation, and even to mix the two, e.g. by convenient use of model replaceability, and top-level variables. To limit the performance loss, equations (not algorithms) were used in event-based models, so as to allow those models to be manipulated with the rest of the simulator. Doing so involves some limitations when porting a pre-existing algorithm into the library, since for example multiple assignments are not allowed. It is the authors' opinion, however, that an accurate translation in the form adopted by the presented library is possible for of any control algorithm one may come across.

For example, the following Modelica code is the event-based implementation of an ISA PID with antiwindup, manual and tracking modes, and bumpless mode switch [17, 18].

```
function Der "This sfunction represents a derivative action"
  input Real sp;
  input Real pv;
  input Real pv_old;
  input Real Td;
  input Real Ts;
  input Real N;
  input Real d_old;
  output Real d;
algorithm
  d := Td/(Td + N*Ts) * d_old - Td*N/(Td + N*Ts) * (pv - pv_old);
end Der;

model Proportional
  RealInput sp "set point";
  RealInput pv "process variable";
  RealOutput p "control signal";
  parameter Real Ts = 0.1 "sample time [s]";
  parameter Real K = 5 "proportional constant";
  parameter Real b = 1 "set point weight";
protected
```

3 Examples

```
  discrete Real sp_d;
  discrete Real pv_d;
  discrete Real p_d(start=0);
equation
  when sample(0,Ts) then
    sp_d = sp;
    pv_d = pv;
    p_d = p;
    p = Pr(pre(sp),pre(pv),K,b);
  end when;
end Proportional;

model Integral
  RealInput sp "set point";
  RealInput pv "process variable";
  RealOutput i "control signal";
  parameter Real Ts = 0.1 "sample time [s]";
  parameter Real Ti = 5 "integral time";
protected
  discrete Real i_d( start=0);
  discrete Real sp_d( start=0);
  discrete Real pv_d( start=0);
equation
  i_d = i;
  when sample(0,Ts) then
    sp_d = sp;
    pv_d = pv;
    i_d = Int(pre(sp),pre(pv),Ti,Ts,pre(i_d));
  end when;
end Integral;

model Derivative
  RealInput sp "set point";
  RealInput pv "process variable";
  RealOutput d "control signal";
  parameter Real Ts = 0.1 "sample time [s]";
  parameter Real Td = 5 "derivative time";
  parameter Real N = 10 "derivative filter";
protected
  discrete Real d_d( start=0);
  discrete Real d_d2( start=0);
  discrete Real pv_d( start=0);
  discrete Real pv_d2( start=0);
  discrete Real sp_d( start=0);
equation
  d_d = d;
  when sample(0,Ts) then
    pv_d = pv;
    sp_d = sp;
    d_d2 = pre(d_d);
    pv_d2 = pre(pv_d);
  end when;
d_d = Der(pre(sp_d),pre(pv_d),pre(pv_d2),Td,Ts,N,pre(d_d2));
end Derivative;

model PID_parallel_AW_Tr_AutoMan
  RealInput sp "set point";
  RealInput pv "process variable";
  RealInput tr "signal followed during the tracking mode";
  RealInput CSman "control signal for manual mode";
  BooleanInput TS "flag for the tracking mode";
  BooleanInput MAN "flag for the manual mode";
  RealOutput cs "control signal";
  Proportional P( Ts=Ts,K=K,b=b) "Proportional block";
  Derivative D( Ts=Ts,Td=Td,N=N) "Derivative block";
  Integral I( Ts=Ts,Ti=Ti) "Integral block";
  parameter Real Ts = 1 "sample time [s]";
  parameter Real Ti = 8 "integral time";
  parameter Real Td = 5 "derivative time";
  parameter Real K = 10 "proportional constant";
  parameter Real b = 1 "weight of the set point in the P action";
  parameter Real N = 10 "derivative filter";
  parameter Real CSmax = 1 "Max cs value";
  parameter Real CSmin = 0 "min Cs value";
protected
  Real control;
equation
  P.sp = sp;
  D.sp = sp;
  I.sp = sp;
  P.pv = pv;
  D.pv = pv;
  I.pv = pv;
  control = if (MAN==false)
    then I.i + P.p + D.d
    else CSman;
  cs = if (TS==true and MAN==false)
    then tr
    else max(CSmin,min(CSmax,control));
end PID_parallel_AW_Tr_AutoMan;
```

We now report two simulation examples. the first is aimed at showing the usefulness of the

possibility of simulating the same regulator as continuous-time and as event-based model, while the second shows a “small but realistic” application of the presented library.

3.1 Example 1

This example refers to some PI/PID control loops, and deals with set point step and ramp responses where the antiwindup mechanism of the regulator comes into play. The process to be controlled is described by the transfer function

$$P(s) = \frac{1}{1 + 2s + s^2/0.016}$$

and the PID regulator

$$R(s) = 10 \left(1 + \frac{1}{30s} + \frac{3s}{1 + 0.3s} \right)$$

is applied to it, in the continuous-time version and as an event-based model with a sampling time of 0.01 s.

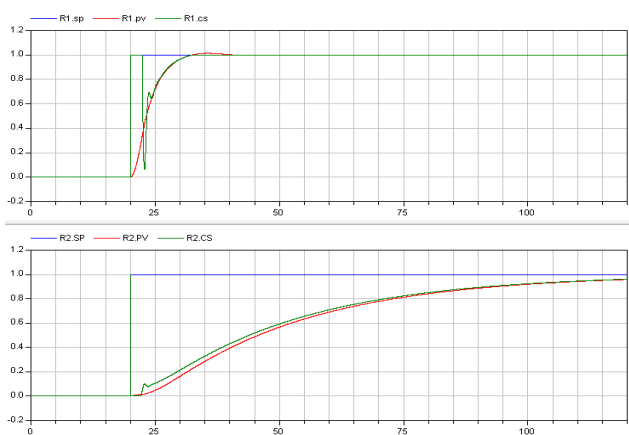


Figure 1: results of example 1.

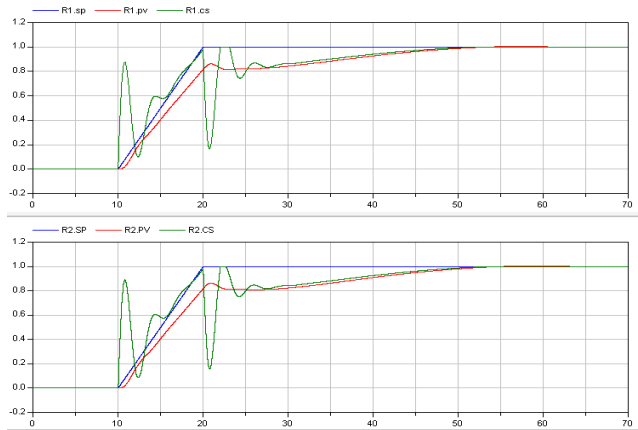


Figure 1 above shows the comparison between the continuous-time (R1) and event-based (R2) controller implementation in the case of a ramp response (left column of plots) and of a step response (right column): SP, PV and CS stand for Set Point, Process (controlled) Variable, and Control Signal, respectively. Apparently, simulating the same controller as a continuous-time or an event-based model (i.e., as it will really be implemented) can give very different results, depending not only on the controller parametrisation, the sampling time and other very well known facts, but also on the control law being incremental or positional, of the antiwindup type, and so on (facts that conversely are frequently overlooked). The example therefore backs up the usefulness of the presented library as far as the control behaviour evaluation is concerned.

3.2 Example 2

This example shows the control of a small manufacturing system where parts are fed to the working area by a conveyor, machined, and then taken away by another conveyor. The detailed sequence of operations is as follows:

- lead one part near the machining area entrance with an input belt,
- push the part into the machining area with an input piston,
- machine the part (drill a hole with a controlled-speed machining head)
- push the part out of the machining area with an output piston,
- and finally lead the part away with an output belt.

The considered machine is synthetically described in figure 2

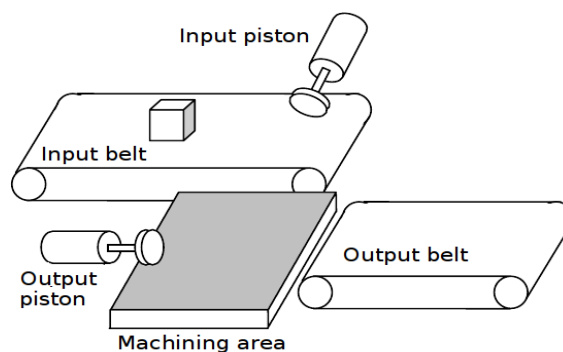


Figure 2: schematic drawing of the machine considered in example 2.

Figure 3 shows the Modelica scheme using some library blocks (mostly set point generators, PIDs, and logic elements), while a sample of simulated transients is given in figure 4.

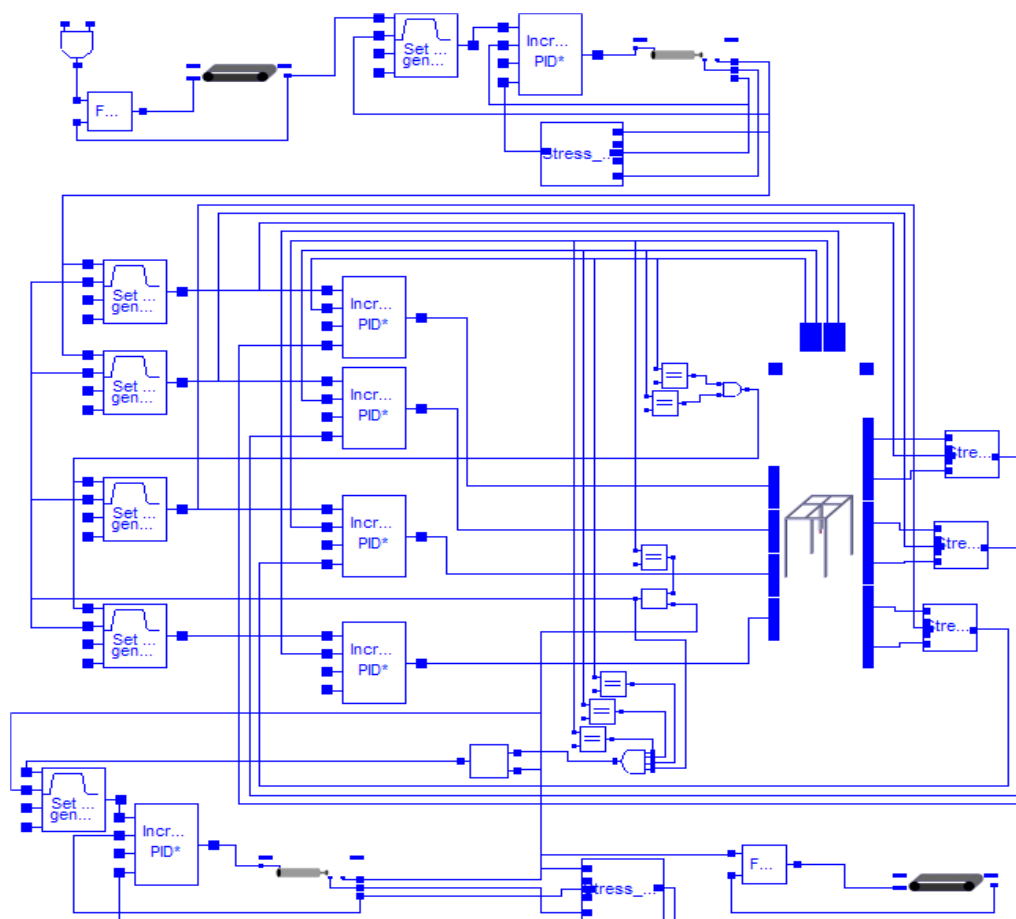


Figure 3: the Modelica scheme using the presented FBD library used in example 2.

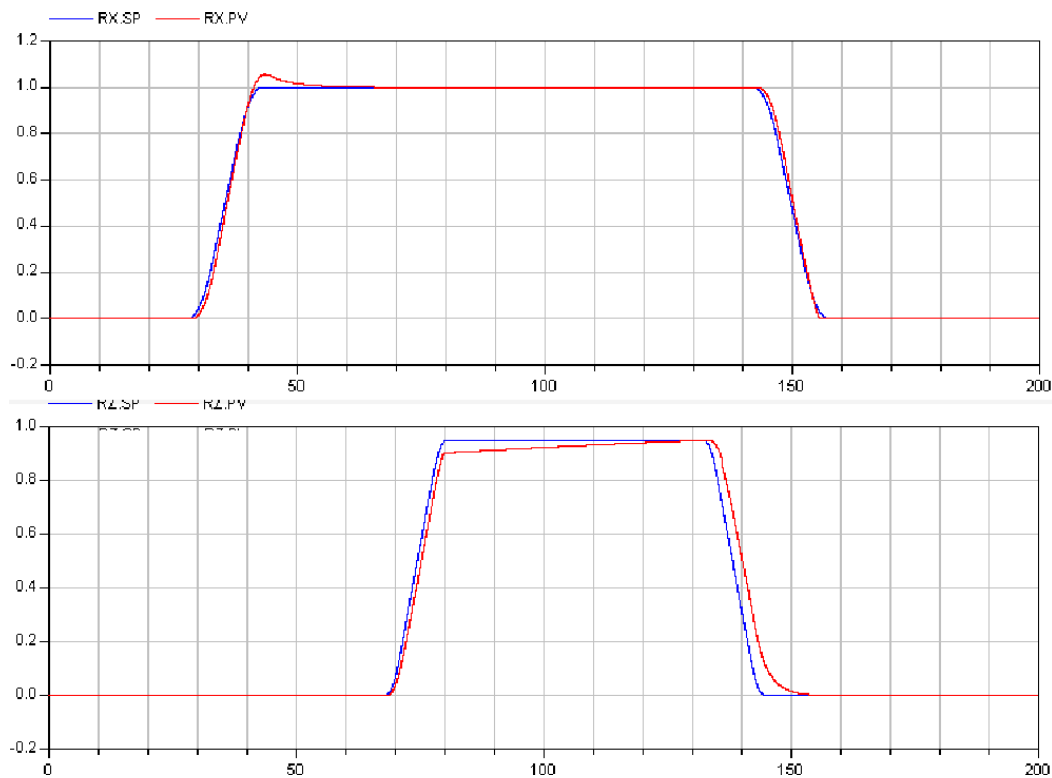


Figure 4: some simulated transients referring to example 2; the upper plot shows the drilling head x position (red) and set point (blue), the lower plot shows the drilling depth (red) and set point (blue).

The similarity of figure 3 with the schemes encountered in many control code development systems are apparent. The example therefore backs up the usefulness of the presented library as far as the clarity of the control specification (in terms of a widely accepted industrial standard) is concerned.

4 Conclusions

A free (GPL) Modelica library for the simulation of logic control systems written in the FBD (Functional Block Diagram) language was presented.

The library adheres to the FBD specifications as defined in the IEC61131.3 standard, and contains not only strictly logic blocks, but also the main types of industrial controllers, particularly of the

PID type. The adoption of an industrial standard facilitates information sharing and greatly reduces ambiguities.

With the presented library, that the user can specify a control system as a continuous-time or an event-based model, for maximum flexibility in fulfilling the simulation needs.

Some simulations were presented to illustrate the usefulness of the library, which will be extended in the future, with respect to both FBD and other IEC-compliant languages.

References

- [1] T. Sato, E. Yoshida, Y. Kakebayashi, J. Asakura, N. Komoda, Application of IEC61131-3 For Semiconductor Processing Equipment, Emerging Technologies and Factory Automation. Proceedings. 2001 8th IEEE International Conference on, 2001.
- [2] J. Huang, Y. Li, W. Luo, X. Liu, K. Nan, The Design of New-Type PLC based on IEC61131-3, Proceeding of the Second Internadonal Conference on Machine Learning and Cybernetics, Xi, 2-5, November 2003.
- [3] D. E. Rivera, M. Morari, and S. Skogestad, Internal model control 4. pid controller design, Ind. Eng. Chem. Res., vol. 25, pp. 252–265, 1986.
- [4] H. Takada, H. Nakata, S. Horiike, A Reusable Object Model for Integrating Design Phases of Plant Systems Engineering, Proceedings of the Fourth International Conference on Computer and Information Technology (CIT'04).
- [5] H. Taruishil, S. Kajiharal, J. Kawamotol, M. Ono, H. Ohtani, Development of Industrial Control Programming Environment Enhanced by Extensible Graphic Symbols, SICE-ICASE International Joint Conference 2006 in Bexco, Busan, Korea, Oct. 18-2 1, 2006.
- [6] Y. Qiliang, X. Jianchun, W. Ping, Water Level Control of Boiler Drum Using One IEC61131-3-Based DCS, Proceedings of the 26th Chinese Control Conference, Zhangjiajie, Hunan, China, July 26-31, 2007.
- [7] M. Bonfé', C. Fantuzzi, L. Poretti, PLC Object-oriented programming using IEC61131-3 norm languages: an application to manufacture machinery, in Proc. of IEEE/ASME Int. Conf. on Advanced Intelligent Mechatronics, vol. 2, pp. 787-792, 2001.
- [8] [Online]. Available <http://www.plcopen.org>.
- [9] J. Roger Folch, J. Pérez, M. Pineda, R. Puche, Graphical Development of Software for Programmable Logic Controllers, 12th International Power Electronics and Motion Control Conference.
- [10] [Misc]. DeltaV: Monitor and control software.
- [11] [Misc]. Labview: <http://www.ni.com/labview>.
- [12] A. Nobuo, I. Kenichi, Y. Eiji, Application portfolios for stardom, 12th International Power Electronics and Motion Control Conference.
- [13] M. Otter, K. E. Årzén, I. Dressler, StateGraph-A Modelica Library for Hierarchical State Machines, 4th International Modelica Conference, March 7-8, 2005.
- [14] O. Johansson, A. Pop, P. Fritzson, Engineering Design Tool Standards and Interfacing Possibilities to Modelica Simulation Tools, 5th International Modelica Conference, September 4-5, 2006.
- [15] E. Tisserant, L. Bessard, M. de Sousa, An Open Source IEC 61131-3 Integrated Development Environment, Industrial Informatics, 5th IEEE International Conference on, 2007.

- [16] [Online]. ISaGRAF:
<http://www.icpdas.com/products/PAC/i-8000/isagraf.htm>
- [17] [book]. O'Dwyer Aidan, Handbook of PI and PID Controller Tuning Rules, Imperial College Press.
- [18] [book]. K. J. Åström and T. Hägglund, Advanced PID control, ISA - The Instrumentation, Systems, and Automation Society, 2005.
- [19] [book]. K. J. Åström and T. Hägglund, PID Control Theory, Design and tuning, ISA, 1995.
- [20] L. Desborough, R. Miller, Increasing customer value of industrial control performance monitoring – Honeywell's experience, Sixth International Conference on Chemical Process Control, AIChE Symposium Series Number 326 (Volume 98), 2002.
- [21] O. Johansson, A. Pop, P. Fritzson, A functionality Coverage Analysis of Industrially used Ontology Languages, in Model Driven Architecture: Foundations and Applications (MDAFA), 2004, 10-11 June, 2004, Linköping, Sweden.

Session 2b

Thermodynamic Systems & Applications

ExternalMedia: A Library for Easy Re-Use of External Fluid Property Code in Modelica

Francesco Casella¹

Christoph Richter²

¹Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy

²Institut für Thermodynamik, TU Braunschweig, Germany
casella@elet.polimi.it

ch.richter@tu-bs.de

Abstract

The modeling of thermo-physical fluid properties is of great importance when modeling thermo-fluid systems. The Modelica Standard Library provides a number of medium models that can be used in component models but are not sufficient in many applications. This paper presents a new interface library with a Modelica front-end that allows for an easy inclusion of external fluid property code in Modelica using the standard interfaces provided in the Modelica.Media library. The new library was developed as an open-source project and is available for free from the Modelica website including an interface to the FluidProp software developed and maintained at TU Delft. The new library can easily be extended to other external fluid property code.

Keywords: external fluid property code; Modelica.Media; thermo-fluid systems

1 Introduction

Modelica is finding more and more applications in the field of thermo-fluid system modeling due to the many advantages of the object-oriented equation-based approach. A fundamental problem in this field is the availability of good Modelica models for the computation of fluid properties. The Modelica.Media library was included in the Modelica Standard Library in version 2.2. It currently provides several ready-to-use models for ideal gases, mixtures, water/steam, moist air, table-based incompressible fluids, and generic linear fluid models which can be used in a wide range of applications. The library and some applications are described in [1] and [2]. However, there exists a large class of engineering systems such as refrigeration systems, heat pumps, or organic Rankine cycles that require accurate models of application-specific two-phase fluids which are currently not provided in the Modelica Standard Library.

One possibility to overcome this limitation is to write the required medium models in Modelica, possibly by conforming to the Modelica.Media interfaces for greater compatibility. The advantage of this approach is that self-contained Modelica models are obtained that can be optimized for efficiency. The major drawbacks are that writing such code requires a sizable investment in terms of time and effort, and that the developed code can only be re-used in a Modelica context.

The other possibility is to take advantage of existing fluid property code developed for general-purpose applications and to interface that code to Modelica. This approach offers a couple of unique advantages compared to a Modelica-internal solution:

- Many existing fluid property codes are well-tested and used in a number of commercial applications and products.
- Many existing fluid property codes provide fast and robust solvers for the inverse iteration of fluid properties.
- External fluid property codes can be used in a number of different software tools such as simulators, office programs, and post-processing tools.

Some existing publications such as [3] show that interfacing external fluid property code from Modelica is a feasible alternative to Modelica-internal solutions. This solution becomes extremely interesting if the effort of developing the interface for any given external fluid property code is kept to a minimum. The ExternalMedia library was developed with this objective in mind. The current implementation considers two-phase, single-substance fluids since this combination already covers many interesting applications that cannot be developed using existing Modelica.Media models. Fluid mixtures might be supported in the future.

The goals of the ExternalMedia library can be summarized as follows:

- The new medium models shall be 100% compatible to the Modelica.Media interface.
- The new interface library that handles all external fluid property codes should work with all available Modelica tools and C/C++ compilers.
- The effort to interface new external fluid property codes should be kept as little as possible.
- The new approach shall be numerically efficient to be comparable with current Modelica-internal solutions.

The new library including all source code will be released on the Modelica website and will be made available under the Modelica license.

2 Architecture of the Library

The new fluid property library consists of three main parts: A Modelica front-end called ExternalMedia, an interface layer written in C, and an object-oriented interface library called ExternalMediaLib written in C++ that handles a number of external fluid property codes.

The Modelica front-end of the new library is the ExternalMedia library, whose class structure is illus-

trated in Figure 1. This Modelica library contains a package named ExternalTwoPhaseMedium that extends from PartialTwoPhaseMedium defined in Modelica.Media.Interfaces. The ExternalTwoPhaseMedium package is generic. The actual external fluid property code used is specified by setting the values of suitable string constants in the medium package. The libraryName specifies the name of the external fluid property code to be used whereas the substanceName defines the name of the substance from this external fluid property code. The mediumName defined in the PartialMedium package in the Modelica.Media library is also passed to the interface library but is not used for the specification of the fluid. The new external medium model can be used in any component model that uses a medium package extending from PartialTwoPhaseMedium.

A set of functions in the ExternalTwoPhaseMedium package corresponds one-to-one to C-functions defined in the C interface layer. These functions are called according to the external function mechanism as defined in the Modelica language specification. The interface layer functions manage a collection of C++ objects that define the interface to the external fluid property codes. A class diagram of this part of the new library is shown in Figure 2.

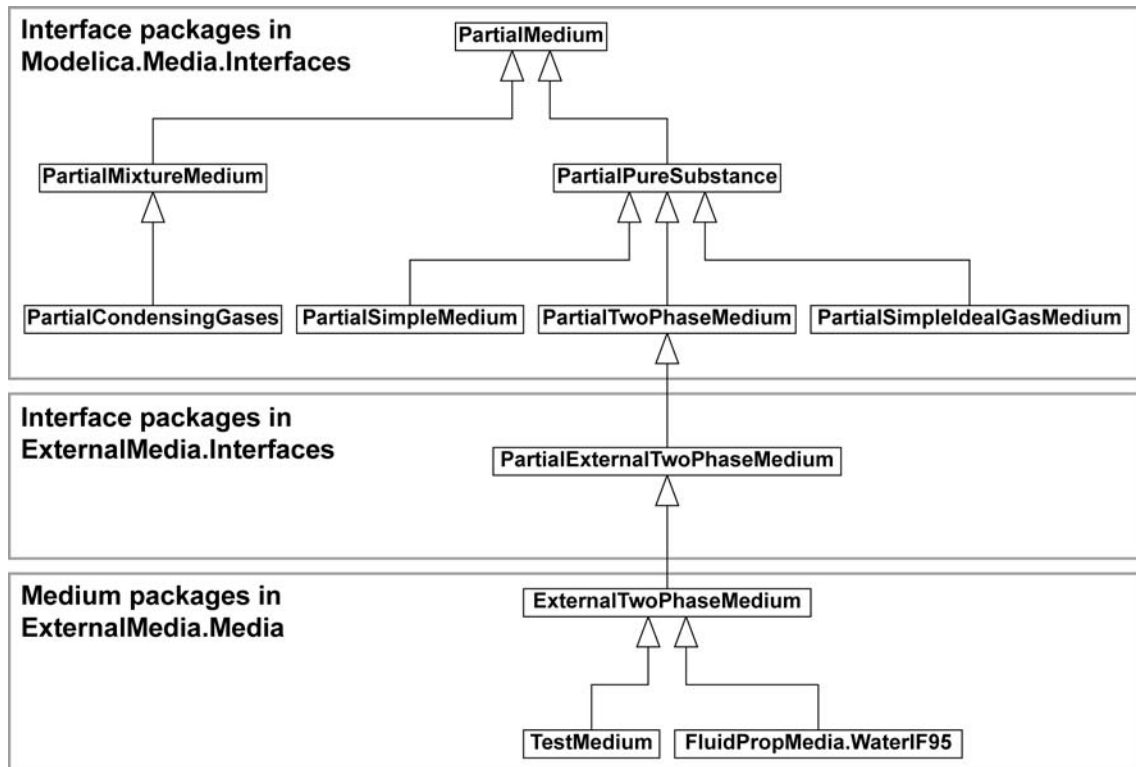


Figure 1: UML class diagram of packages in Modelica.Media and ExternalMedia library.

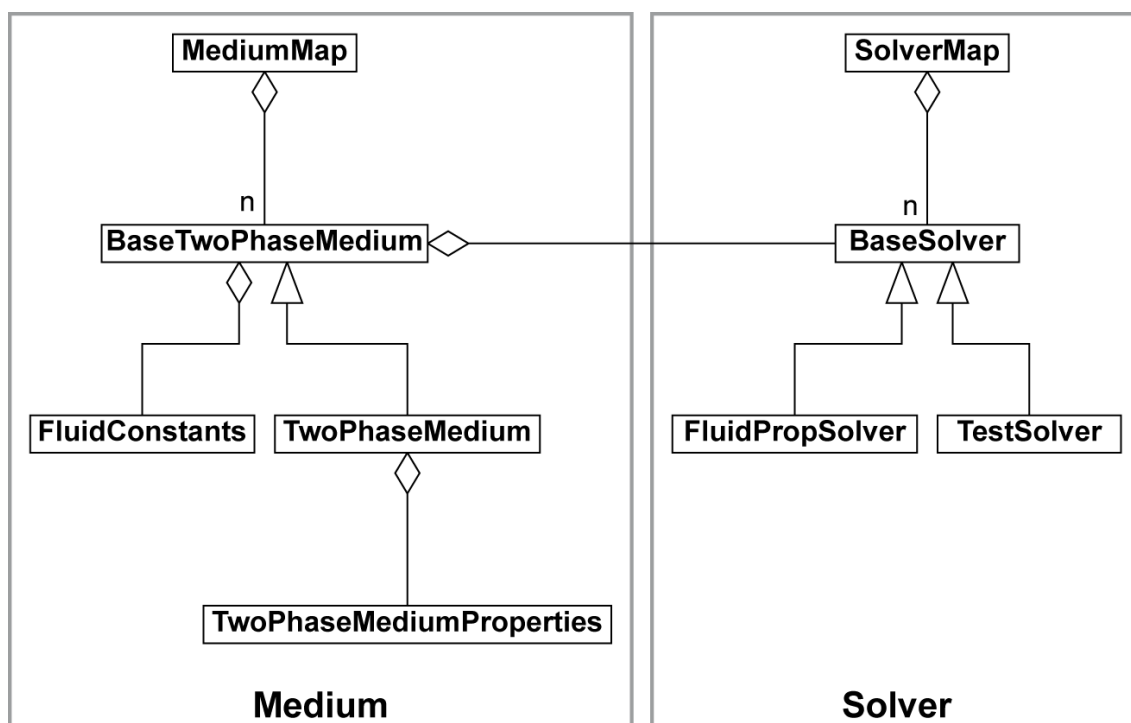


Figure 2: UML class diagram of C++ objects in the ExternalMediaLib library.

The first fundamental object is the Solver object that encapsulates the external fluid property code. In order to manage several different solvers at the same time, the interface layer defines the map SolverMap which is a collection of Solver objects indexed by the strings defined in the ExternalTwoPhaseMedium package. Each time an external function is called, these strings are passed as arguments. This allows for an instantiation of the corresponding solver when the function is called the first time and for the interface layer to point to the correct solver in any subsequent function call.

The second fundamental object is the TwoPhaseMedium object which corresponds with a point in a thermodynamic phase diagram such as a pressure-enthalpy diagram or a point on the saturation curve for saturation properties. Each TwoPhaseMedium object contains a pointer to the corresponding Solver object and a record of type TwoPhaseMediumProperties which is used as a cache record containing all possible thermodynamic properties including transport properties. All instances of these objects are stored in the map MediumMap which is indexed by an integer called uniqueID.

In order to understand how the library works, consider the following code snippet:

```
import SI = Modelica.SIunits;

package Toluene
  extends ExternalTwoPhaseMedium(
    mediumName="Toluene",
    libraryName="REFPROP",
    substanceName="Toluene");
end Toluene;

model Example
  Toluene.ThermodynamicState state;
  SI.Density d;
  SI.SpecificEnthalpy h;
equation
  state = Toluene.setState_pT(1e5,300);
  d = Toluene.density(state);
  h = Toluene.specificEnthalpy(state);
end Example
```

The setState_pT() function of the medium package calls the corresponding C function of the interface layer, passing the values of pressure and temperature as well as the three medium identification strings. If those strings are not already present in the SolverMap, an instance of the corresponding solver (in this case REFPROP [4]) is added to the SolverMap. Subsequently, an instance of TwoPhaseMedium is added to the MediumMap and the setState_pT() function of the Solver is called to compute all fluid properties. The computed fluid properties are stored in the TwoPhaseMediumProperties object that acts as a cache record. Finally, a unique identification number is returned to identify the TwoPhaseMedium object in the MediumMap. This number is stored in the ThermodynamicState record together with the values of pressure, temperature, density, specific enthalpy, and

specific entropy. Note, that the `setState_pT()` function in Modelica is an impure function since it returns a different `uniqueID` each time it is called.

When the `density()` function is called, the corresponding interface layer function is called with the `uniqueID` stored in the `ThermodynamicState` record. This allows for retrieving the already computed value for the density from the `TwoPhaseMedium` object in the `MediumMap`. The same thing happens when the `specificEnthalpy()` function is.

At the next simulation step, when the `setState_pT()` function is called again, a new `TwoPhaseMedium` object is allocated in the `MediumMap` and a new unique identification number is returned. In order to avoid running out of memory, the `MediumMap` is used as a circular buffer with a predefined maximum number of `TwoPhaseMedium` objects. The size of the buffer must be large enough to accommodate all `setState_XX()` function calls during a single simulation step.

The most straightforward implementation of the Solver objects computes all possible fluid properties at once when a `setState_XX()` function is called. This often is a reasonable option since most of the CPU time is often spent on the inverse iteration while the additional cost of computing all fluid properties is small. However, it is always possible to decide that the `setState_XX()` functions of the Solver only compute and store some of the properties and that the additional computations are triggered when the functions to retrieve these additional properties are called. This is very often a good idea for the transport properties. This mechanism allows for avoiding unnecessarily repeated computations in a flexible way that is 100% compatible with the existing structure of the `Modelica.Media` package.

Note, that the call of the `setState_XX()` functions will usually be performed before the other function calls because of the BLT partitioning of equations performed by the Modelica compiler. If this does not happen (e.g., due to the presence of implicit equations), the property functions could be called before the unique identification number has been set, thus with a default `uniqueID=0`. In this case it is still possible for the interface layer to select the correct solver by using the medium identification strings and to compute the required property using the values of pressure, temperature, etc. stored in the `ThermodynamicState` record. The `uniqueID` argument is thus introduced for efficiency reasons, i.e. to avoid unnecessarily repeated computations, but is not required for the correctness of the results.

If the `BaseProperties` model defined in the `Modelica.Media` library is used to compute the medium

properties, the circular buffer for the `MediumMap` can be avoided. A unique identification number is instead stored in each instance of the `BaseProperties` model. This number is set once and for all during the initialization phase. The `setState_XX()` functions are then called within the `BaseProperties` model by explicitly supplying the `uniqueID`. The same `TwoPhaseMedium` object in the `MediumMap` is thus used for all computations in the corresponding `BaseProperties` object. In order for the `MediumMap` to distinguish between these static unique identification numbers and the transient unique identification numbers discussed in the previous paragraphs, the former are given positive values while the latter ones are given negative numbers.

3 Implementing new Medium Models

Implementing the interface to a new external fluid property code is a straightforward task requiring a limited amount of time.

First of all a new Solver must be defined, extending from the `BaseSolver`. This new Solver has to implement all abstract `setState_XX()` functions defined in the base class. These functions will actually call the external fluid property code and store the retrieved properties in the `TwoPhaseMediumProperties` cache.

Then, a few lines of code must be added to the `getSolver()` function of the `SolverMap` object in order to recognize the new identification strings of the additional external fluid property code. All remaining functionality is already provided by the library framework.

4 Current Status and Future Development

The framework of the new library for the support of external two-phase single-substance medium models is complete. Two Solvers are already implemented. The first Solver, `TestSolver`, is a dummy fluid model roughly corresponding to cold water which can be used to troubleshoot the C/C++ and Modelica compiler setup without worrying about the actual external code. It can also be used as a starting point for new user-defined fluid property codes.

The second available Solver is an interface to the `FluidProp` software [5] developed and maintained at TU Delft which provides a common interface to several external fluid property codes including `StanMix`, `TPSI`, and the whole `REFPROP` database. `FluidProp` can be downloaded for free even though the REF-

PROP module requires purchasing a license from NIST. Since FluidProp is based on the proprietary COM architecture by Microsoft, the corresponding solver can only be compiled under MS Windows using a MS Visual Studio compiler, even though an extension based on open-source architectures is envisioned for the near future.

The library framework is fully compliant with standard Modelica (2.2 and 3.0) and with standard ANSI C/C++. New Solvers can thus be implemented and used within any Modelica tool, using any C/C++ compiler.

The library, including all source code, will be released under the Modelica License and will be made available on the Modelica website. The C/C++ source code is fully documented, using the Doxygen tool. Future development might include the development of new general-purpose Solvers as well as the development of an external media interface for fluid mixtures.

Furthermore, the object-based fluid property library TILFluids developed at TU Braunschweig and presented in [6] uses the code of the presented external fluid property library and provides a different Modelica interface. TILFluids also provides interfaces to other software tools such as MS Excel or MATLAB/Simulink that might be included in a future release of the ExternalMedia library.

5 Conclusions

This paper presents a new fluid property library for two-phase single-substance fluids that allows for an easy inclusion of external fluid property code in Modelica, using the standard interfaces for two-phase media defined in the Modelica.Media library. Any model designed to use models derived from these standard interfaces can therefore be used without any modification. The new library is freely available under the Modelica license, and can easily be extended by including other external fluid property codes. Further development might extend the interface to single-phase pure substances and mixture media, as well as two-phase mixture media. Interested users are welcome to use the new library in their applications and are invited to contact the authors for contributions to the project.

References

- [1] H. Elmqvist, H. Tummescheit, and M. Otter. *Object-Oriented Modeling of Thermo-Fluid*

Systems. In Proc. of 3rd International Modelica Conference, pages 269-286, Linköping, November 2003.

- [2] F. Casella, M. Otter, K. Prölb, C. Richter, and H. Tummescheit. *The Modelica Fluid and Media library for modeling of incompressible and compressible thermo-fluid pipe networks*. In Proc. of 5th International Modelica Conference, pages 631-640, Vienna, September 2006.
- [3] H. Tummescheit and J. Eborn. *Chemical Reaction Modeling with ThermoFluid/MF and MultiFlash*. In Proc. of 2nd International Modelica Conference, pages 31-39, Oberpfaffenhofen, March 2002.
- [4] E. W. Lemmon, M. Huber, and M. McLinden. *NIST Standard Reference Database 23: Reference Fluid Thermodynamic and Transport Properties-REFPROP, Version 8.0*. National Institute of Standards and Technology, Standard Reference Data Program, Gaithersburg, 2007.
- [5] *FluidProp: A software for the calculation of thermophysical properties of fluids*. <http://fluidprop.tudelft.nl/>
- [6] C. Richter. *Proposal of New Object-Oriented Model Libraries for Thermodynamic Systems*. Doktorarbeit, TU Braunschweig, to be published in 2008.

ThermoBondLib – A New Modelica Library for Modeling Convective Flows

François E. Cellier
ETH Zürich
Switzerland
FCellier@Inf.ETHZ.CH

Jürgen Greifeneder
Technische Universität Kaiserslautern
Germany
Greifeneder@EIT.Uni-KI.DE

Abstract

This paper describes a new library designed for modeling convective flows in physical systems. The library is based on bond graph technology. Thermo-bonds are introduced as a means to capture the convective flow of the internal energy of matter through a physical system. ThermoBondLib is a companion library to the BondLib and MultiBondLib libraries that were released in 2005 and 2006, respectively.

Keywords: *Bond Graph, Thermo-bond Graph, Convective Flow*

1 Introduction

1.1 Introduction to Thermo-bond Graphs

Bond graphs [1] describe the flow of power through a physical system. Each power flow can be written as the product of two variables, one extensive, the other intensive. For example, electrical power can be written as the product of voltage and current, whereas mechanical translational power can be written as the product of force and velocity.

Since all physical systems have to comply with energy conservation laws, a tool that balances all power flows in a physical system can, in principle, be used to model any such system.

A Modelica library based on bond graph technology, BondLib [2], was released in 2005 and presented at the 4th Modelica conference in Hamburg-Harburg. The library won the 1st prize for a free Modelica library at that conference.

Whereas bond graphs are capable of describing all types of physical systems, it may not be convenient to do so. Bond graph models are rather primitive, low-level descriptions of a physical system, and therefore, a bond graph representing a complex

physical system will necessarily be large and poorly readable.

However, bond graphs are nevertheless very useful, because they represent the lowermost graphical interface to a physical system model that is still fully object-oriented. Thus, by wrapping other, higher-level modeling methodologies around a bond graph implementation, the semantic distance between the lowermost graphical layer and the bottom equation layer can be reduced. This makes a library that is based on bond graph technology easily maintainable.

BondLib contains several wrapped sub-libraries for modeling particular classes of physical systems. For modeling electronic analog circuits, BondLib offers a complete implementation of Spice built on bond graph technology [3]. For modeling 1D mechanical systems, BondLib offers two sub-libraries, one for translational, the other for rotational motion, that are similar in nature to the corresponding sub-libraries of the Modelica standard library, but are built upon bond graph technology. For thermal systems, BondLib offers a heat transfer sub-library, etc.

Since its inception, BondLib has been updated several times to include new wrapped sub-libraries. For example, the mechanical and thermal sub-libraries became available only in 2007.

However, it may still be too inconvenient to model a complex physical system down to the bond graph layer in a single step. For example, the semantic distance between a multi-body system description of a 3D mechanical system and a bond graph description thereof is still too large. Another intermediate layer is needed to be inserted between these two layers.

To this end, a second bond graph library, MultiBondLib [5], was released in 2006 and presented at the 5th Modelica conference in Vienna. The library

won the 1st price for a free Modelica library at that conference.

When dealing with 2D or 3D mechanical systems, the d'Alembert principle needs to be formulated several times, once for each spatial direction. Also, the principle needs to be formulated separately for translational and rotational motions. This calls for a vector representation of bonds, which is precisely the framework that MultiBondLib has been based upon.

MultiBondLib also offers several wrapped libraries for higher-level descriptions of 2D and 3D mechanical systems. These higher-level descriptions are internally implemented as multi-bond graphs. The multi-bond graphs are then directly translated to the equation layer using the matrix/vector notation of Modelica. This was simpler and more efficient than translating the multi-bond graphs first down to regular bond graphs in a graphical fashion.

When dealing with convective flows, there is yet another complication to be considered [4-7]. When considering a mass moving macroscopically from one place to another, that mass carries along with it its internal energy of matter, U :

$$U = T \cdot S - p \cdot V + g \cdot M \tag{1}$$

where T denotes temperature, S is the entropy, p represents pressure, V stands for the volume, g symbolizes the Gibbs potential (specific enthalpy), and M finally captures the mass.

The flow of internal energy, $Udot = dU/dt$, can be written as:

$$Udot = T \cdot Sdot - p \cdot q + g \cdot Mdot \tag{2}$$

where $q = Vdot$ represents the volumetric flow rate. Consequently, a mass flow is always accompanied by a heat flow and a volumetric flow.

For this reason, mapping convective flows directly down to regular bond graphs is once again cumbersome. The size of these bond graphs would grow too fast.

The ThermoBondLib library presented in this paper provides another vector bond graph representation, whereby each thermo-bond is composed of three regular bonds, one representing mass flow, a second representing volumetric flow, and a third representing heat flow. A thermo-bond can be envis-

aged as a parallel connection of three regular bonds, as shown in Fig.1:

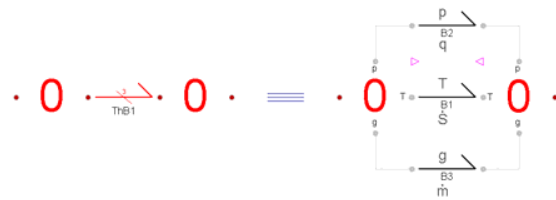


Figure 1: Representation of a thermo-bond

For reasons of efficiency, the thermo-bond model has not been composed in ThermoBondLib in a graphical fashion, but rather by equations directly. Hence the bond graph of Fig.1 offers only a conceptual decomposition of a thermo-bond.

1.2 The Thermo-bond Connectors

The thermo-bond connectors would, in analogy to the regular bond connectors have to carry at least seven variables: the three effort variables, T , p , and g ; the three flow variables, $Sdot$, q , and $Mdot$; and finally, the directional variable, d , that assumes a value of $d = -1$ at the connector at which the bond emanates and a value of $d = +1$ at the connector to which the bond leads.

In reality, the thermo-bond connector is an 11-tuple. It also carries (for convenience) the three state variables, S , V , and M , and in addition a Boolean variable, $Exist$, that is set true when there is mass to be transported, and is set false when the mass is close to zero. The Boolean variable is useful, because the models often operate on specific flows, i.e., flows per unit of mass, and Modelica becomes agitated when we attempt to divide a flow rate by zero.

The thermo-bond connector is depicted in Fig.2.

```
connector ThBondCon "Bi-directional thermo-bond graph connector"
  Modelica.SIunits.Temperature T "Temperature";
  Modelica.SIunits.Pressure p "Pressure";
  Modelica.SIunits.SpecificEnthalpy g "Gibbs potential";
  Modelica.SIunits.ThermalConductance Sdot "Entropy flow";
  Modelica.SIunits.VolumeFlowRate q "Volume flow";
  Modelica.SIunits.MassFlowRate Mdot "Mass flow";
  Modelica.SIunits.Entropy S "Entropy";
  Modelica.SIunits.Volume V "Volume";
  Modelica.SIunits.Mass M "Mass";
  Real d "Directional variable";
  Boolean Exist "True if substance exists";
end ThBondCon;
```

Figure 2: Thermo-bond connector

The measurement unit of entropy flow is currently set to *ThermalConductance* rather than the dimensionally compatible *EntropyFlowRate*, because the

latter unit is still missing in the Modelica Standard Library.

Just like in the case of the regular bond connectors, the thermo-bond connectors come in three varieties, one used for a-causal thermo-bonds, and the other two used for the two types of causal thermo-bonds.

1.3 Advantages of Bond Graph Modeling

Why is it useful to represent component models of a Modelica library internally by means of bond graphs?

To demonstrate the usefulness of this approach to modeling physical systems, let us consider the model of a heat conduction element offered as a component model of the heat transfer sub-library of the Modelica standard library. The model is shown in Fig.3.

```

model ThermalConductor
  "Lumped thermal element transporting heat without storing it"
  extends Interfaces.Element1D;
  parameter SI.ThermalConductance G "Constant thermal conductance of material";
equation
  Q_flow = G*dT;
end ThermalConductor;
    
```

Figure 3: Conduction model of Standard Library

The thermal conductor is modeled in the same way as a regular electrical resistor. Unfortunately, this model is incorrect.

To demonstrate the problems with this model, let us look at the bond graph representation of a resistor, as shown in Fig.4.

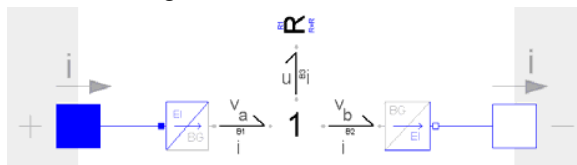


Figure 4: Bond graph representation of a resistor

The electrical power leaving the positive terminal equals $v_a \cdot i$, whereas the power arriving at the negative terminal equals $v_b \cdot i$. At the 1-junction, the potential drop is being calculated, and the difference in power, $u \cdot i$, gets sent to the resistor.

What happens to the power as it arrives at the resistor? It cannot vanish. It gets dissipated into heat.

In an electrical resistor model, it may make sense to ignore (exclude from the model) the thermal phenomena. However in a thermal resistor, this makes no sense whatsoever.

Thus, the dissipated heat (generated entropy) needs to be routed back into the thermal network.

Hence a correct model of the thermal conductor would have to look as depicted in Fig.5:

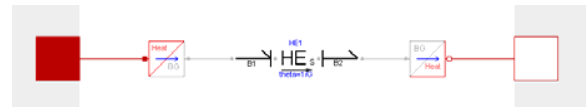


Figure 5: Bond graph model of thermal conductor

where the heat exchanger, HE, is internally represented as shown in Fig.6.

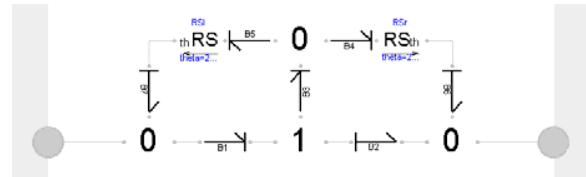


Figure 6: Bond graph model of heat exchanger

The generated heat is divided in two at the top 0-junction, and the two halves of the generated entropy flows are rerouted to the nearest 0-junctions to the left and the right of the 1-junction.

Using the bond graph approach, the shortcomings of the model contained in the Modelica Standard Library became obvious at once.

2 Potential Equilibration

In convective flows, each of the three sub-flows can equilibrate its potential variable separately.

When two bodies are in contact with each other, heat diffusion takes place between them. Heat diffusion is modeled by the thermal conductor discussed earlier. However in the new model, the two connectors to the left and the right are now thermo-bond connectors. The enhanced heat exchanger model is shown in Fig.7.

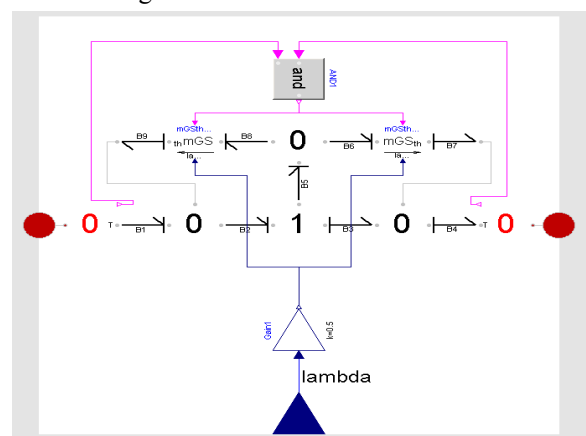


Figure 7: Thermo-bond graph model of heat exchanger

The resistive source, RS, was replaced by a conductive source, GS. The thermal conduction, λ , is imported as a modulating signal, rather than being treated as a parameter value, and finally, heat conduction only takes place if both neighboring substances exist.

The red 0-junctions are special junctions that take the red vector bond apart and make the individual component flows available.

The second type of potential equilibration is the volume work, represented in ThermoBondLib by the pressure/volume exchanger, PVE. The pressures in two neighboring volumes will equilibrate, if the two volumes are separated by a movable membrane. The corresponding model is shown in Fig.8.

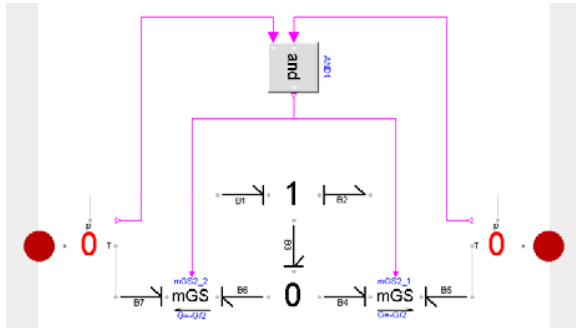


Figure 8: Thermo-bond graph model of volume work

Volume work is a dissipative phenomenon. In the process of pressure equilibration, entropy is being generated that is fed back into the thermal ports of the neighboring 0-junctions.

The conductance values of the volume work element are negative, because a positive pressure difference leads to a negative volume flow: if the pressure on the left side is larger than that on the right side, the membrane is pushed to the left larger and not smaller.

The two bonds to the left and the right of the conductive sources are pointing in opposite direction, because a negative volume flow leads to a positive entropy flow in accordance with Eq.(2).

The third type of potential equilibration is the mixing element, represented in ThermoBondLib by the mass exchanger, ME. The Gibbs potentials in two neighboring volumes may equilibrate, if the two masses are able to mix. The corresponding model is shown in Fig.9.

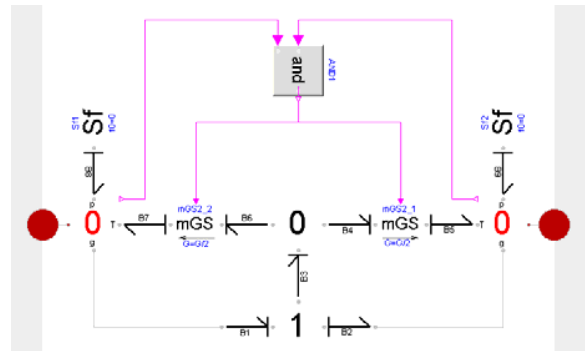


Figure 9: Thermo-bond graph model of mixing

Also mixing is a dissipative phenomenon. In the process of mixing two fluids, mixing entropy is being generated.

The ME model is a bit more problematic than the other two potential equilibration models, because mass cannot really flow without taking its volume and heat along. The model can be used to explain the cause of mixing entropy, but in practice, more complex models will be needed in most cases to describe mixing phenomena.

The three potential equilibration elements have in common that they don't require state information. All three elements equilibrate one potential only, i.e., they don't lead to associated flows of the other two types. All three phenomena are dissipative in nature, i.e., generate entropy in the process of potential equilibration.

3 The Substance Models

Whereas the transport models, such as the potential equilibration models, are responsible for computing the flows in and out of volumes, the substance models compute the three potentials of a volume. They also compute the three state variables by means of integrating over the difference between inflows and outflows. In bond graph terminology, the substance models are capacitive fields, CF.

Unfortunately, the substance models cannot be coded in a fully substance-independent fashion. Separate models need to be created for different types of substances. Of course, these models have a lot in common with each other, and inheritance schemes can be set up that restrict the recoding between similar models to the bare minimum, but this hasn't been accomplished yet. At the current time, substance models have only been made available for air, water, and water vapor.

The air model shall serve here as an example. It is depicted in Fig.10. The model is fully coded by means of equations. It computes the three potentials, T , p , and g , and determines the values of the three state variables, S , V , and M , by integrating the three flow variables, $Sdot$, q , and $Mdot$.

```

model Air "Capacitive field representing air"
  extends Interfaces.PassiveOnePort;
  parameter Modelica.SIunits.Entropy S0=6.81010184 "Entropy if no air";
  parameter Modelica.SIunits.Volume V0=0.83112221e-3 "Volume if no air";
  parameter Modelica.SIunits.Mass M0=1e-3 "Mass if no air";
  parameter Modelica.SIunits.SpecificHeatCapacity cp=1004.0
    "Heat capacity of air at constant pressure";
  parameter Modelica.SIunits.SpecificHeatCapacity R=287.2 "Gas constant";
  parameter Modelica.SIunits.Mass epsM=0.5e-6
    "Smallest mass distinguishable from zero";
  parameter Boolean fict=false "True is fictitious values are used";
  parameter Modelica.SIunits.Temperature T_fict=298.53
    "Fictitious temperature is no air";
  parameter Modelica.SIunits.Pressure p_fict=1e5
    "Fictitious pressure if no air";
  Modelica.SIunits.Entropy S_int "Entropy of air";
  Modelica.SIunits.Volume V_int "Volume of air";
  Modelica.SIunits.Mass M_int "Mass of air";
  Modelica.SIunits.SpecificHeatCapacity cv
    "Heat capacity of air at constant volume";
  Modelica.SIunits.SpecificVolume v "Specific volume";
  Modelica.SIunits.SpecificEntropy s "Specific entropy";
  Real ln_v "Natural logarithm of specific volume";
equation
  der(M_int) = Mdot;
  der(S_int) = Sdot;
  der(V_int) = q;
  Exist = M_int > epsM;
  cv = cp - R;
  v = if Exist then V_int/M_int else 0;
  s = if Exist then S_int/M_int else 0;
  ln_v = Modelica.Math.log(V_int/M_int);
  p = if Exist or not fict then T*R*M_int/V_int else p_fict;
  T = if Exist or not fict then 293.15*exp((s - 6813.7 - R*(ln_v + 0.17245))
    /cv) else T_fict;
  g = T*(cp - s);
  M = if Exist then M_int else M0;
  V = if Exist then V_int else V0;
  S = if Exist then S_int else S0;
end Air;

```

Figure 10: Model of air storage

Since air can be considered an ideal gas, the pressure may be computed from the ideal gas equation. All fluids are subject to an *equation of state*, and that equation is being used to determine the pressure.

All fluids are also subject to a *caloric equation of state*. That equation is being used to determine the temperature. Once the temperature and the specific entropy are known, the Gibbs potential can be determined as well.

Additional code has been added to prevent the model from dividing by zero in case the volume gets completely emptied out.

4 Evaporation and Condensation

A second class of transport phenomena beside from potential equilibration is the phase change of a fluid. Water can boil off (evaporate), and it can condensate out, either in the bulk or on a cold surface.

Whereas some transport phenomena are dissipative, like the potential equilibration phenomena, oth-

ers are reversible. Evaporation and condensation are reversible transport phenomena. In the process of evaporation, the activation energy is taken out of the thermal domain. Sensible heat gets converted to latent heat. However in the reverse process of condensation, the previously borrowed latent heat gets converted back to sensible heat¹.

How much evaporation/condensation takes place, i.e., where the flow equilibrium is between the liquid and the gaseous forms of a fluid, is determined by minimizing the overall energy of the system. However in practice, this energy minimization problem is hardly ever solved on-line. The ThermoBondLib code, like most other such programs, computes the amount of evaporation/condensation using steam tables.

5 The Pressure Cooker

We are now able to code a first example using the ThermoBondLib library. To this end, we shall model a pressure cooker.

Inside the pressure cooker, there are three types of substances: water, air, and water vapor. As the pressure cooker is being heated, more water boils off, producing additional water vapor. The water vapor would like to occupy more space than the water in its liquid form, but it cannot, because the total volume of the pressure cooker is fixed. Consequently, the pressure values inside the pressure cooker are rising. Fig.11 depicts a preliminary model representing the pressure cooker.

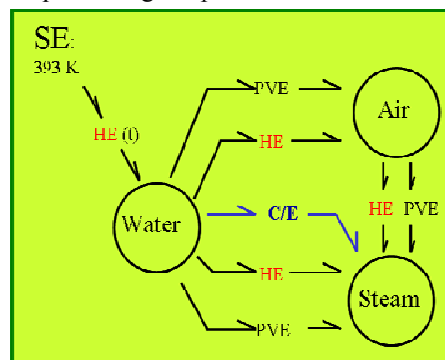


Figure 11: Basic pressure cooker model

There are three different volumes containing three different substances. Potential equilibration in the form of heat exchange and volume work is al-

¹ In the classical model, sensible heat is taken out of the liquid phase during evaporation, but gets added to the gaseous phase during condensation.

lowed to occur. Furthermore, there is evaporation and condensation taking place between the water volume and the steam volume.

An improved model takes into account that heating and cooling take place not only through the bottom of the pot, but also through the metal walls, i.e., we need to add boundary layers representing the air and the steam in the vicinity of the metal walls. The enhanced model is shown in Fig.12.

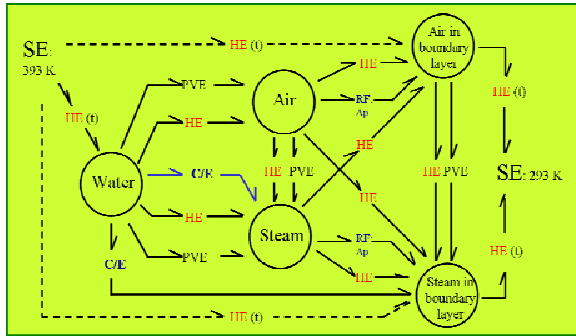


Figure 12: Enhanced pressure cooker model

Special volume work elements are used between the bulk and the two boundary layers. These models ensure that the volume of the boundary layers doesn't change.

We are now ready to encode the model using ThermoBondLib, as shown in Fig.13.

There are five separate 0-junctions representing the five different substances. In some cases, individual 0-junctions had to be split into two or even three 0-junctions to provide for a sufficiently large number of connectors.

At each of these 0-junctions, a capacitive field is attached, modeling the properties of the five substances. Between the 0-junctions, there are placed all the transport models representing the exchange of mass, volume, and heat between the five substances.

Whereas this model is still fairly simple, it already occupies an entire screen. Model wrapping techniques ought to be used to represent the model at a yet higher level, below which the thermo-bond graphs can be hidden.

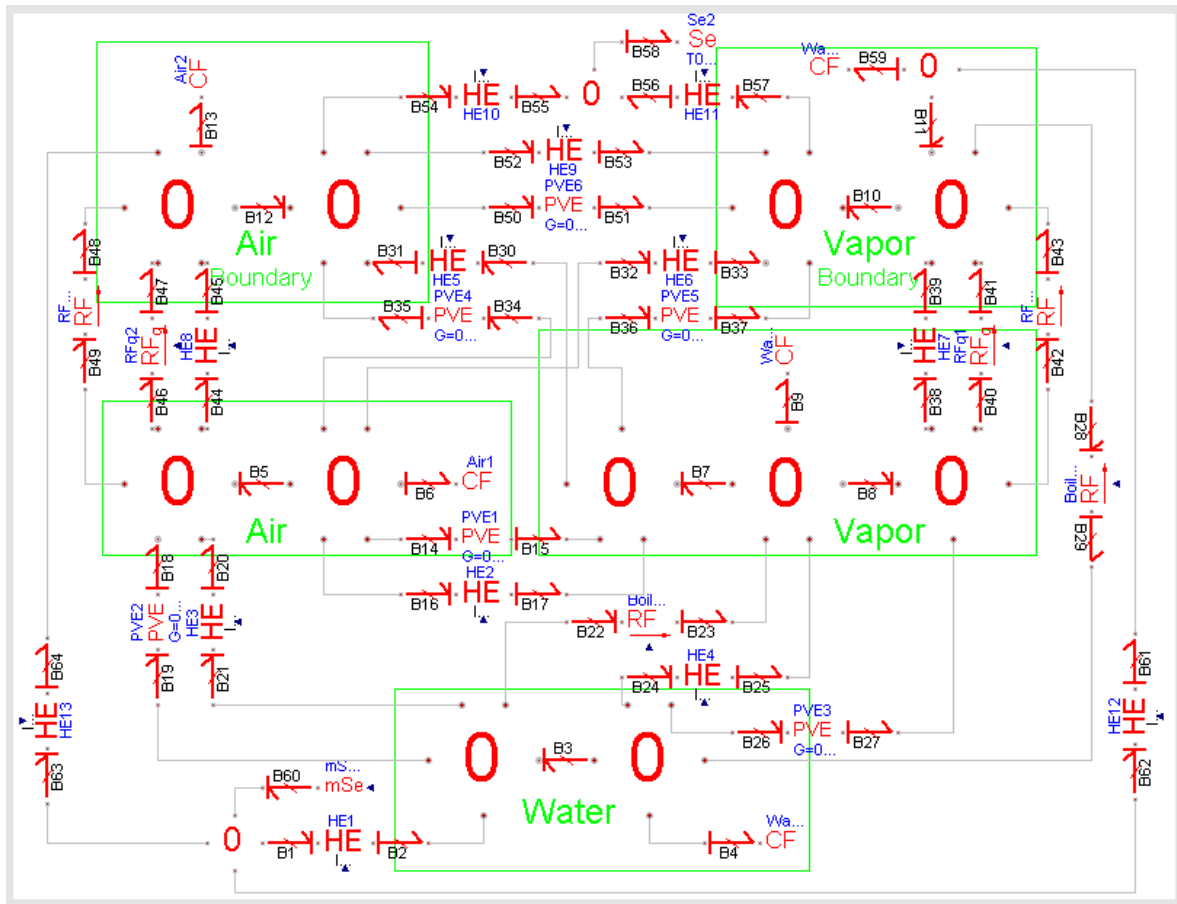


Figure 13: Coded pressure cooker model

Such a wrapped model might represent the control volumes as containers, and the transport models between them as pipe segments. Yet, the wrapping of thermo-bond graphs representing convective flows hasn't been accomplished yet. Wrapped sub-libraries shall be added to ThermoBondLib at a later time.

Some simulation results are shown in Figs.14 and 15. The pressure cooker is placed on a hot surface at time 0. Cold water is poured over it after 10,000 seconds. Whereas the five temperature values are significantly different, the pressure values are almost indistinguishable.

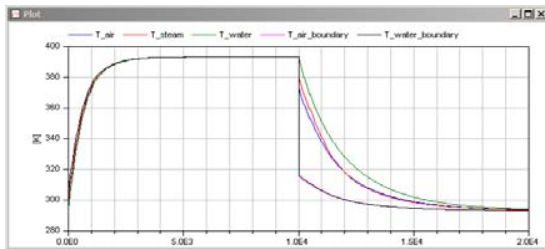


Figure 14: Temperature values of pressure cooker

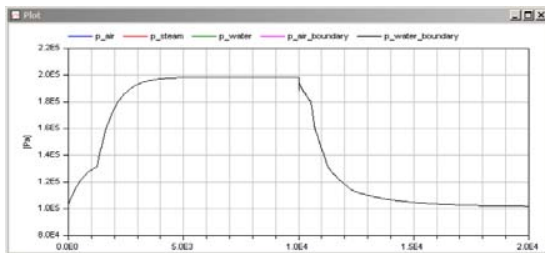


Figure 15: Pressure values of pressure cooker

After approximately 1300 seconds, the water begins to boil, which leads to a knee in the pressure curves.

6 The Air Balloon

As a second example, we shall consider a bottle containing compressed air, from which an air balloon is to be filled.

Here, we are facing a new complication. If we were to model the bottle containing compressed air by means of the same capacitive field as before, the volume of the bottle would shrink as air is allowed to escape from it. Hence we need to create a new capacitive field that models the storage of air under conditions of constant volume. This model is shown in Fig.16.

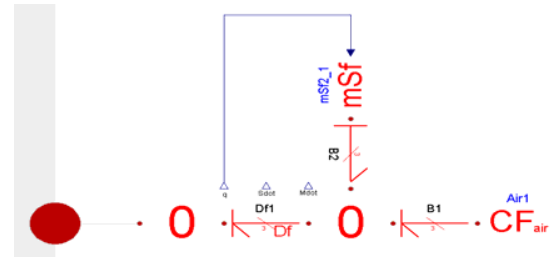


Figure 16: Air storage under isochoric conditions

The isochoric air storage model is built hierarchically from a regular air storage model. The volumetric flow coming out of the storage is measured using a flow sensor, Df. A counter-flow of equal magnitude is generated by the modulated flow source, mSf. It takes its energy from the thermal domain. Hence the bottle keeps a constant volume. Instead of shrinking in size, the bottle cools down.

We are now ready to model the air balloon system. The model is shown in Fig.17.

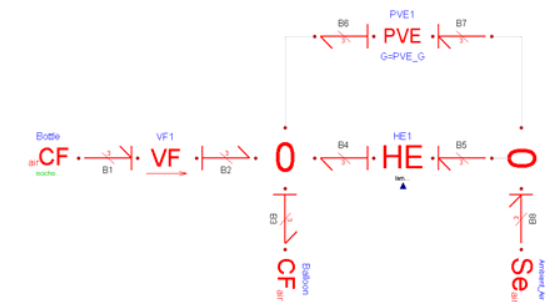


Figure 17: Air balloon model

The CF element to the left of Fig.17 represents the bottle. It is modeled using the isochoric air storage element of Fig.16. The CF element in the center represents the balloon. It is modeled using the regular air storage element of Fig.10.

The effort source, Se, to the right represents the ambient air. Between the balloon and the ambient air, potential equilibration (heat exchange and volume work) are allowed to take place.

The transport of a fluid across a pipe satisfies the wave equation. When cutting the pipe into individual segments, each segment can be modeled using a container, a capacitive field, representing the compressibility of the fluid, and using a transporter, an inductive field, representing the inertia of moving mass.

In the model of Fig.17, the volume flow element, VF, represents the transporter. A volume flow is obtained that is caused by the pressure difference between the two connectors. The volume flow induces a proportional mass flow and a proportional heat flow.

The volume flow element is depicted in Fig.18. The top portion of the bond graph represents the volumetric flow. The volumetric flow is being computed inside the inductor, I. A flow sensor, Df, measures the flow and induces proportional mass and entropy flows using internal flow sources, mSf. The center portion of the bond graph represents the entropy (heat) flow portion of the model, whereas the lowermost part calculates the mass flow.

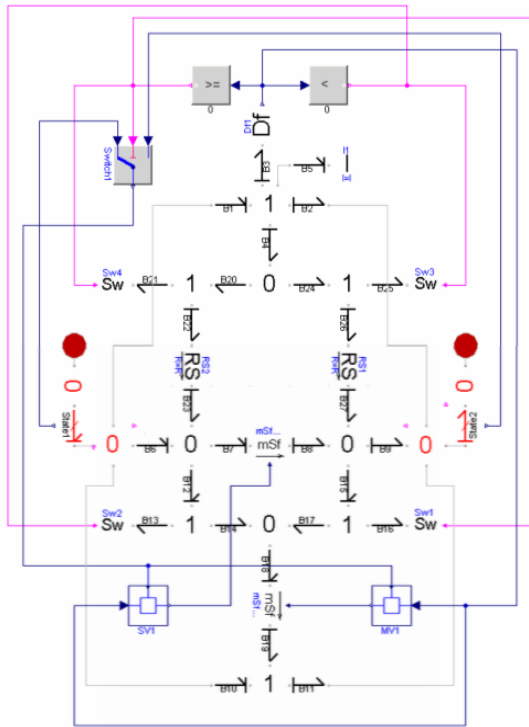


Figure 18: Inductive field representing transporter

As is to be expected, the temperature, pressure, and mass in the bottle decrease, whereas the volume remains constant. In contrast, mass and volume in the balloon increase, whereas temperature and pressure remain almost constant.

Some simulation results are shown in Figs.19-21. Fig.19 shows the temperature values in the bottle and the balloon; Fig.20 depicts the volumes of the two storages; and Fig.21 presents the two air masses.

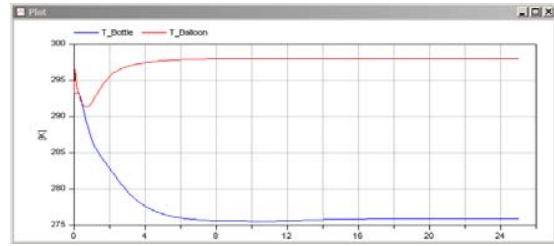


Figure 19: Temperature values of bottle and balloon

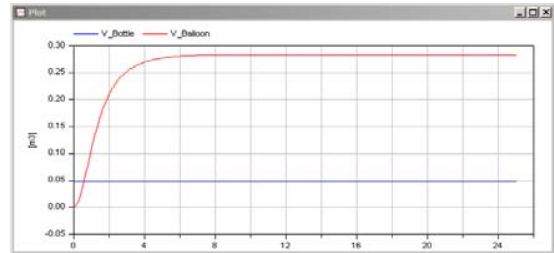


Figure 20: Volumes of bottle and balloon

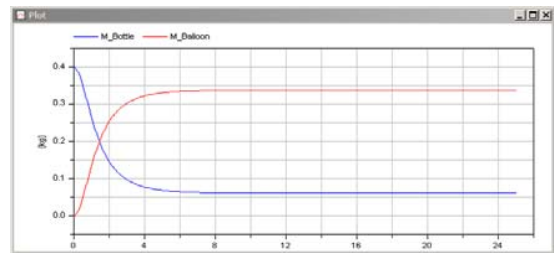


Figure 21: Air masses of bottle and balloon

The temperature in the balloon changes only temporarily, whereas the bottle cools down significantly. The volume of the balloon grows, whereas that of the bottle remains constant. Finally, mass is being transported from the bottle to the balloon. The sum of masses in the two containers remains constant.

7 The Water Loop

As a third example, we shall consider a closed water loop. Such water loops are found frequently in heating systems. They may represent either water that circulates between a water heater and a set of radiators; alternatively, such a loop may represent water (or glycol) circulating between a water heater and a set of (thermal) solar collectors; and finally, it may represent glycol circulating between a heat pump and the well of a geothermal system.

The water loop is represented in the model by four pipe segments and four water storages. The

water storages are modeled as isochoric capacitive fields. Three of the four pipe segments are specified as inductive fields, whereas the fourth one represents the pump.

The overall model is shown in Fig.22. The pipe segment containing the pump, depicted in the model as a forced volume flow, FVF, is built up internally in essentially the same way as the free volume flow, VF, used to represent regular pipe segments. The only difference is that the inductor is eliminated from the model and replaced by an external (regular) bond connector at which the mechanical pump can be connected. The pump itself is represented by a small DC motor.

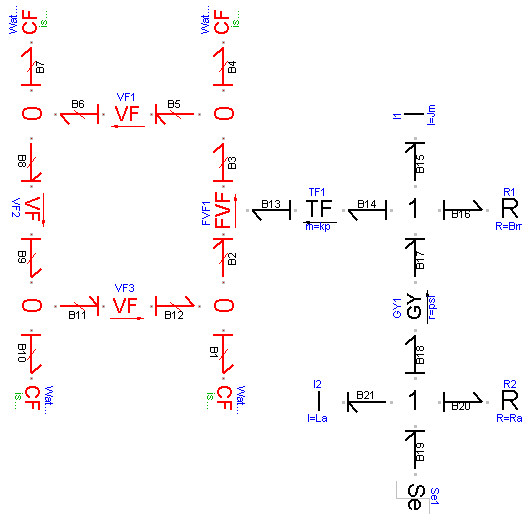


Figure 22: Water-loop model

The (black) effort source at the lower right corner of Fig.22 represents the armature voltage. The I and R elements next to it model the armature inductance and the armature resistance. The GY element higher up describes the transformation of electrical to mechanical (rotational) power. The I and R elements yet higher up represent the inertia of the rotor and the friction of the mechanical bearings. The TF element to their left transforms mechanical into hydraulic power.

The black (right) portion of Fig.22 is a regular bond graph, modeled using BondLib, whereas the red (left) portion of Fig.22 is a thermo-bond graph, modeled using ThermoBondLib.

Some simulation results are shown in Figs.23 and 24. Fig.23 shows the pressure values in the four water storages. The pump generates a high pressure value at its exit that is then successively reduced again by the subsequent regular pipe segments. The highest pressure is p_2 , the pressure of

the capacitive field immediately above (i.e., at the exit of) the pump, whereas the lowest pressure is p_1 , the pressure of the capacitive field below (i.e., at the entrance of) the pump. After the pump is turned on, the pressure values oscillate for a little while, before they stabilize at new values.

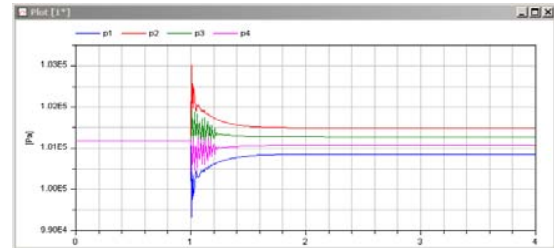


Figure 23: Pressure values in the four water storages

Fig.24 shows the mass flows in the three regular pipe segments. They are virtually indistinguishable one from another. At time 1 sec, the pump is switched on. It takes roughly 1 sec for the mass flow to fully build up.

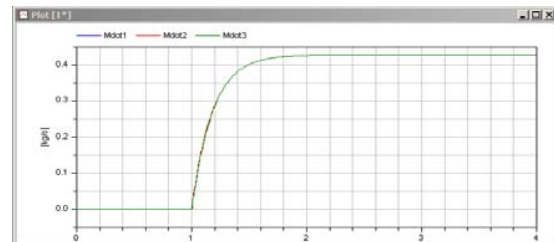


Figure 24: Mass flows in pipe segments

The three examples presented in this paper are included in the ThermoBondLib library, a free Modelica library downloadable from the Modelica website.

8 Conclusions

In this paper, a new free Modelica library for modeling and simulating convective flows in physical systems has been introduced.

ThermoBondLib has been designed as a graphical modeling library based on the thermo-bond graph methodology. Thermo-bond graphs were formally introduced by Greifeneder in his Diploma thesis [4].

Contrary to other types of systems, convective flows cannot be modeled in a completely abstract fashion. Different storages (capacitive fields) will look slightly different one from another. At the current time, storage (substance) models have only been made available for air, water, and water

vapor. Urgently needed are storage models for different types of glycol as well as for different types of industrial oils.

Although thermo-bond graphs are quite well readable, once a user has gotten familiarized with the methodology, bond graphs in general offer a fairly low-level interface to modeling. Bond graphs offer the simplest (lowest-level) interface to modeling physical systems that is still fully object-oriented. Hence wrapped thermo-bond graphs are more suitable for the description of complex systems involving convective flows. Wrapped thermo-bond graph sub-libraries shall be added to ThermoBondLib in due course.

Finally, thermo-bond graphs can also be used for the thermodynamic description of chemical reaction systems. First thermo-bond graph models of a few chemical reaction systems (hydrogen bromide and ammonia synthesis) were successfully coded in [4]. However, these models have not yet been added to the officially released version of ThermoBondLib.

References

- [1] Cellier, F.E.: [Continuous System Modeling](#). Springer-Verlag, New York, 1991
- [2] Cellier, F.E., Nebot, A.: [The Modelica Bond Graph Library](#). In: *Proceedings of the 4th International Modelica Conference*, Hamburg-Harburg, Germany (2005) Vol. 1, 57-65
- [3] Cellier, F.E., Clauß, C., Urquía, A.: [Electronic Circuit Modeling and Simulation in Modelica](#). In: *Proceedings of the Sixth Eurosim Congress on Modelling and Simulation*, Ljubljana, Slovenia (2007) Vol. 2, 1-10
- [4] Greifeneder, J.: [Modellierung thermodynamischer Phänomene mittels Bondgraphen](#). Diploma Thesis, University of Stuttgart, Germany, 2001
- [5] Greifeneder, J., Cellier, F.E.: [Modeling Convective Flows Using Bond Graphs](#). In: *Proceedings of ICBGM'01, 5th SCS Intl. Conf. on Bond Graph Modeling and Simulation*, Phoenix, Arizona (2001) 276-284
- [6] Greifeneder, J., Cellier, F.E.: [Modeling Multiphase Systems Using Bond Graphs](#). In: *Proceedings of ICBGM'01, 5th SCS Intl. Conf. on Bond Graph Modeling and Simulation*, Phoenix, Arizona (2001) 285-291
- [7] Greifeneder, J., Cellier, F.E.: [Modeling Multi-element Systems Using Bond Graphs](#). In: *Proceedings of ESS'01, 13th European Simulation Symposium*, Marseille, France (2001) 758-766
- [8] Zimmer, D., Cellier, F.E.: [The Modelica Multi-bond Graph Library](#). In: *Proceedings of the 5th International Modelica Conference*, Vienna, Austria (2006) Vol. 2, 559-568



François E. Cellier received his BS degree in electrical engineering in 1972, his MS degree in automatic control in 1973, and his PhD degree in technical sciences in 1979, all from the Swiss Federal Institute of Technology (ETH) Zurich. Dr. Cellier worked at the University of Arizona as professor of Electrical and Computer Engineering from 1984 until 2005. He recently returned to his home country of Switzerland. Dr. Cellier's main scientific interests concern modeling and simulation methodologies, and the design of advanced software systems for simulation, computer aided modeling, and computer-aided design. Dr. Cellier has authored or co-authored more than 200 technical publications, and he has edited several books. He published a textbook on Continuous System Modeling in 1991 and a second textbook on Continuous System Simulation in 2006, both with Springer-Verlag, New York.



Jürgen Greifeneder received a diploma degree in Engineering Cybernetics from the University Stuttgart, Germany in 2002. Dr. Greifeneder then switched to the University of Kaiserslautern, where he received a Ph.D. in Electrical and Computer Engineering in 2007 with a dissertation on the formal analysis of temporal behavior of Networked Automation Systems (NAS) by use of probabilistic model checking. Scientific stays at the University of Arizona (USA), at the Ecole Normale Supérieure de Cachan (F), and at the Universidade de Brasília (BR) completed his education. Dr. Greifeneder's primary research interests concern modeling, simulation, and control methodologies.

FluidDissipation - A Centralised Library for Modelling of Heat Transfer and Pressure Loss

Thorben Vahlenkamp Stefan Wischhusen
XRG Simulation GmbH,
Harburger Schloßstraße 6-12, 21079 Hamburg
{vahlenkamp, wischhusen}@xrg-simulation.de

Abstract

A new Modelica library centralising heat transfer and pressure loss calculations of energy systems called FluidDissipation will be presented. The goal of the library is to deliver a broad range of heat transfer and pressure loss correlations independent of the thermo-hydraulic framework and easy to implement (functional approach) for industrial use. Concept and numerical challenges of the library development will be described as well as first applications (Pipe flow; Environmental control system).

The FluidDissipation library is developed within the European research project Eurosyslib-D of ITEA2 funded by the Federal Ministry of Education and Research (BMBF) for 30 month started in November 2007.

Keywords: heat transfer; pressure loss; simulation; dissipation

1 Introduction

Energy conversion in any thermo-hydraulic process [1] is declined due to unwanted heat transfer (as a result of temperature difference) and pressure losses (as a result of friction) of a working fluid. Both physical phenomena increase entropy and decrease exergy of an energy system. Therefore the amount of energy of a working fluid to be transformed into mechanical work is dissipated.

These fluid dissipation effects (e.g. pressure loss of pipe network) have to be compensated by higher energy supply of other system components (e.g. delivery height of pumps). A reduction of fluid dissipation effects is a way to optimise efficiency of a thermo-hydraulic process with a corresponding minimisation of operation costs. Thus modelling fluid dissipation effects are necessary for thermo-hydraulic processes

to evaluate existing energy systems and to find out optimising potentials.

Therefore the target of the FluidDissipation library is to deliver a centralised open source Modelica library including verified and validated correlations describing heat transfer and pressure losses of fluids for energy systems. Applications of the FluidDissipation library (e.g. incompressible pipe flow; Air conditioning heat exchanger with compressible moist air) will be developed with the use of the Modelica.Fluid library.

2 Library concept

The main goal of FluidDissipation as an open source library is to allow the usage of dissipation models in every thermo-hydraulic framework. Also the FluidDissipation library can be used as a multi domain base library to achieve a maximum of flexibility in implementation and further application to energy systems. The way to obtain an overall use of the FluidDissipation library is to build up the library according to the following implementation methods:

- Library development with functional approach (literally use of function calls)
- Input and output arguments of function calls delivered by records (like geometric parameters and fluid properties)
- Implementation of continuous functions for efficient numerical simulation

3 Numerical aspects of transient fluid flow modelling

In literature there are a lot of heat transfer and pressure loss correlations within restricted boundary conditions. In order to get dissipation functions applicable

for a broad region of fluid conditions every restricted mathematical description has to be numerically improved for efficient simulation. Numerical improvement of dissipation functions will be verified by the authors under the following aspects:

- Enlargement of heat transfer and pressure loss functions with restricted boundary conditions to a broader region via numerical interpolation with respect to physical correctness
- Use of pressure loss functions in dependence of functional output targets like:
 - Mass flow rate for compressible fluid flow or
 - Pressure loss for incompressible fluid flow
- Inverting of documented pressure loss functions for compressible fluid flow according to mathematical feasibility
- Linearisation of pressure loss functions for compressible fluid flow at small mass flow rates and reverse flow to avoid numerical difficulties
- Usage of inline integration [2] to improve numerical behaviour (if supported by modelling software)

4 Implementation

The concept of the FluidDissipation library allows both the interoperability with other thermo-hydraulic framework as well as an easy implementation intended for further industrial use as a result of the **functional approach (using literally function calls with records for input/output arguments)**.

The principle of the easy to use implementation for a new base pipe model is pointed out in Figure 1 to Figure 3. In Figure 1 the structure to build a new base pipe model is shown as example for this implementation. The new base pipe model consists of the following components of a chosen thermo-hydraulic library (Modelica.Fluid):

- Hydraulic and thermal connectors for data exchange
- Control volume for calculation of thermodynamic state
- Medium model (e.g. Modelica.Media) for calculation of fluid properties

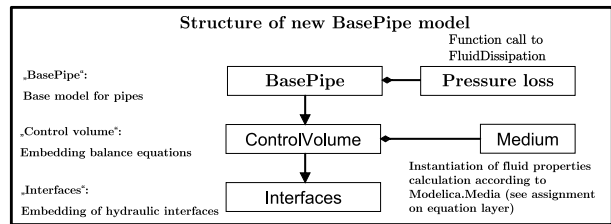


Figure 1: Implementation of a new base pipe model out of FluidDissipation functions - Structure

According to the proposed structure in Figure 1 the user needs to concentrate only on the following steps to implement the missing dissipation calculations successfully. In the following the new base pipe is modelled adiabatic and the further implementation is explained with respect to pressure losses.

1. Create a new model with inherited hydraulic and thermal connectors of chosen thermo-hydraulic library
2. Inheritance of a control volume
3. Instantiation of the medium model
4. Add corresponding records of dissipation calculation on diagram layer of model (see Figure 2)
5. Assign record variables to input and output arguments of chosen function in equation layer of model (see Figure 3)

The diagram layer of this implementation of pressure loss with the used records is shown in Figure 2.

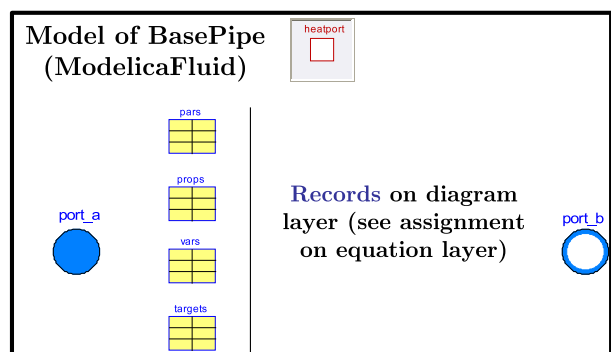


Figure 2: Implementation of a new base pipe model out of FluidDissipation functions - Diagram layer

Finally the pressure loss variables of the records in the diagram layer have to be assigned to the pressure loss function in the equation layer according to Figure 3. In this example an inline function for the mass flow rate

is used. The advantage of an inline function is that either the mass flow rate or the pressure loss can be calculated in dependence of desired target variable in the current model. The medium variables are calculated for a design flow direction from the upstream control volume. Therefore the designed flow direction has to be ensured by the thermo-hydraulic process itself.

```

// Record and pressure drop function assignment
// Fluid properties
props.vis = vis;
props.rho = medium.d;

// Input variables (Here: pressure loss vars.dp)
vars.dp = port_a.p - port_b.p;
vars.m_flow = targets.M_FLOW;

// Output variables
targets.DP = vars.dp;
// Calculate mass flow rate out of pressure difference
targets.M_FLOW =
FluidDissipation.PressureLosses.OnePhase.General.Generic
PressureDropDensity_M_FLOW_Inline(vars.dp, props, pars);
    
```

Figure 3: Implementation of a new base pipe model out of FluidDissipation functions - Equation layer

5 Example application of FluidDissipation

5.1 Pressure loss in straight pipes

Straight pipes are one of the most frequently used devices in the modelling of an thermo-hydraulic process. Therefore also the development of the FluidDissipation library starts with the modelling of pressure loss in a straight pipe using the functional approach for implementation. The result of the pressure loss calculation in a straight pipe in terms of the Darcy friction factor λ_D is shown in Figure 4. The pressure loss calculation for straight pipe flow is based on the following basic correlations:

$$\Delta p = \underbrace{\zeta_{tot}}_{\text{Total pressure loss coefficient}} \cdot \frac{\rho \cdot u^2}{2} \quad (1)$$

$$\zeta_{tot} = \underbrace{\zeta_{fri}}_{\text{Frictional pressure loss}} + \underbrace{\zeta_{loc}}_{\text{Local pressure loss}} \quad (2)$$

$$\zeta_{fri} = \underbrace{\lambda_D}_{\text{Darcy friction factor}} \cdot \frac{L}{d_h} \quad (3)$$

The typical behaviour of the Darcy friction factor λ_D in a straight pipe can be divided into three flow regimes

in dependence of the Reynolds number Re and relative roughness k with the following behaviour:

- Laminar regime (I) at small Reynolds number (Hagen-Poiseuille equation):

λ_D independent of k

Decrease of λ_D with increasing Re

- Transition regime (II) for Reynolds number in between 2300 to 4000 (Smoothing function):

λ_D slightly dependent of k

Increase / decrease of λ_D with increasing Re

- Turbulent regime (III) at high Reynolds number (Numerical Colebrook-White equation):

Increasing λ_D with increasing k

λ_D independent of Re

For very high absolute roughnesses (average height of asperities inside pipe) an additional numerical improvement has to be done for the calculation of the pressure loss. In Figure 4 the laminar regime (I) is calculated independent of the roughness. Nevertheless a numerical abortion of the solver occurs if the absolute roughness of a straight pipe is very large. In this case the difference from the end of the laminar regime to the start of the turbulent regime is not stable even if a smoothing function is used. A numerical improvement of this problem is found in [3] with the modelling of λ_D from the Hagen-Poiseuilles calculated in dependence of the relative roughness k according to Samoilenko with an variable end of the laminar regime and corresponding Reynolds number $Re_{laminar}^{end}$:

$$Re_{laminar}^{end} = 754 \cdot \exp\left(\frac{0.0065}{k}\right) \quad (4)$$

The end of Hagen-Poiseuilles law in dependence of relative roughness leads to an variable upper boundary for laminar fluid flow and a better numerical stability. This numerical improvement is based on the physical behaviour shown in corresponding measurements according to [3] for commercial tubes and it is now implemented in the FluidDissipation library (e.g. bends).

5.2 Simple environmental control system

The second example demonstrates the feasibility of using heat transfer and pressure loss functions from the FluidDissipation library for system simulation. In Figure 5 a simple environmental control system of an aircraft is modelled.

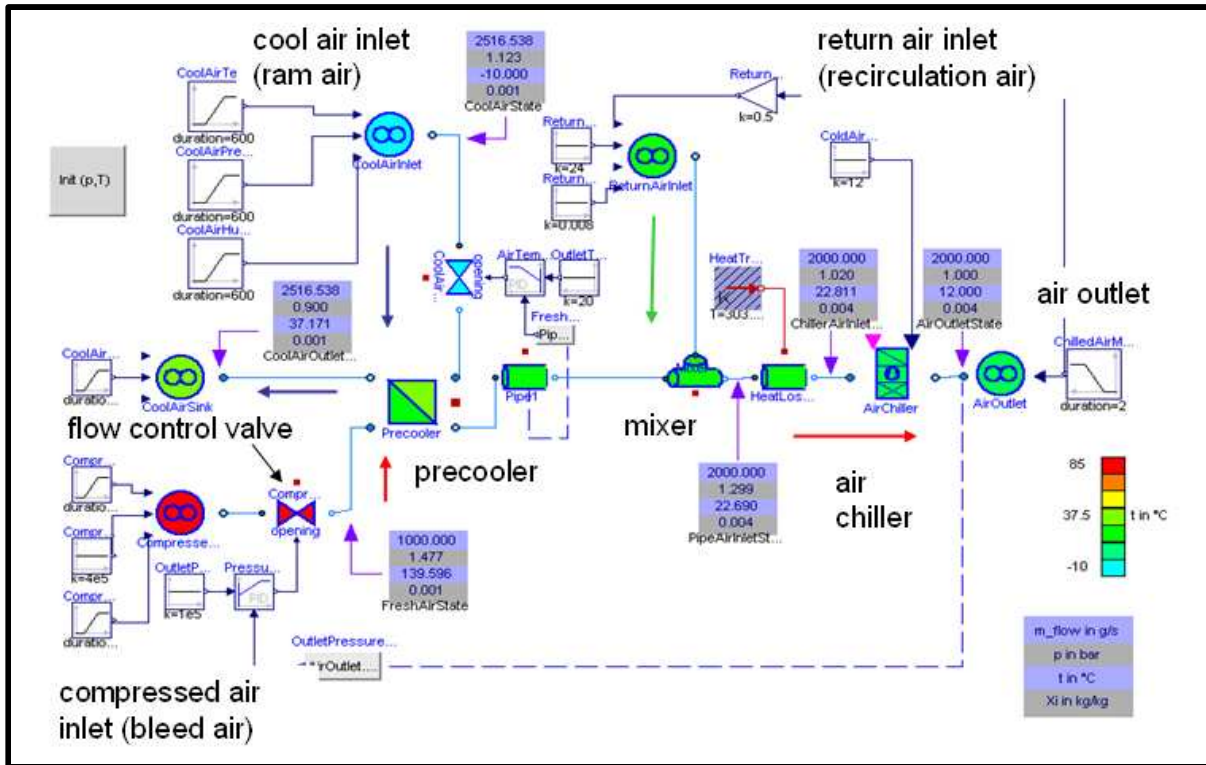


Figure 5: Model of a simple environmental control system for aircrafts

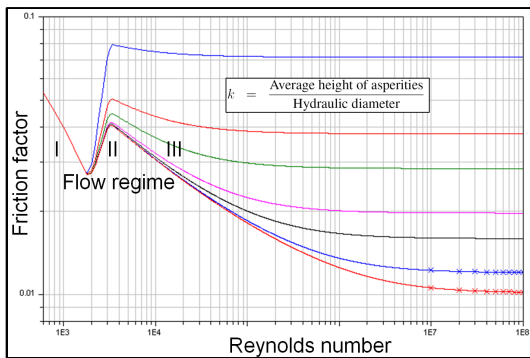


Figure 4: Darcy friction factor λ_D in dependence of Reynolds number Re with relative roughness k as parameter

The simple example of an environmental control system for an aircraft consists of the following features:

- Varying ambient conditions according to flight or ground case of aircraft
- Moist air of Modelica.Media library used as medium
- Compressed fresh air (bleed air) is precooled by ambient air (ram air)
- 50% of hot cabin air is recirculated and mixed with precooled a fresh air

- Air chiller required for temperature control of inlet air for cabin
- Flow control valve for pressure control of inlet air for cabin
- **All models apply FluidDissipation heat transfer and pressure loss functions**

The boundary conditions for a flight test with the simple environmental control system (ECS) in Figure 5 are listed in Table 1. A flow diagram for the main part of the ECS-model is shown in Figure 6. The aim of the shown ECS is to deliver 2 kg/s of moist air with an inlet temperature of 12°C and an ambient pressure of 1 bar with the temperature and pressure control through chiller and control valves.

The most important results of a flight test under the boundary conditions listed in Table 1 are commented in the following. According to the high pressure of 2 bar of the bleed air out of a turbine from an aircraft the flow control valve has to adjust the pressure loss to achieve the desired inlet pressure of 1 bar to the cabin in the ground case. The pressure loss function inside the flow control valve adjusts the needed pressure loss via opening. In Figure 7 the transient pressure of compressed bleed air is shown from the inlet and achieves ambient condition after the chiller.

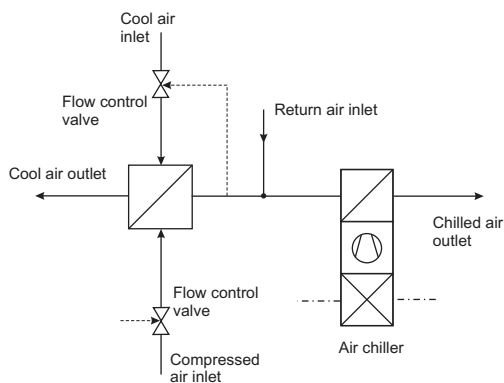


Figure 6: Example air conditioning system layout

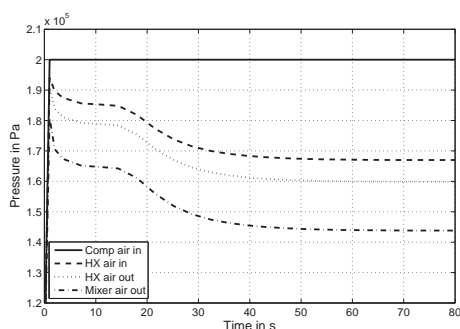


Figure 7: Transient pressures of compressed air from inlet to mixer

The pressure control for the cabin is realised due to the relative opening of the flow control valve and corresponding pressure loss according to Figure 8. The total pressure loss coefficient ζ_{tot} is increasing as result of a decreasing opening of the flow control valve (higher local pressure losses ζ_{loc}).

Also the effect of heat transfer losses to the environment have been considered in a straight pipe in between the mixer and chiller unit with a konstant heat transfer coefficient. Finally in Figure 5.3 the inlet temperature of moist air to the cabin as main task of the environmental control system is simulated. It is shown that the existing environmental control system of the aircraft under ground conditions is not able to fulfill the default inlet temperature of 12°C to the cabin. To reach the set temperature for the ground case the cooling capacity of the chiller has to be raised. However during the flight case the ram air reaches the set value of 12°C for inlet temperature due to the decreasing ambient pressure and temperature leading to higher precooling. Therefore all models applying pressure loss calculation (e.g. Flow control valve, precooler

Table 1: Boundary conditions for simulation

Compressed air inlet			
p	bar	2.0	
ϑ	°C	140	
X_i	kg/kg	0.012 ... 0.001	
Cool air inlet			
p	bar	1.5 ... 1.2	
ϑ	°C	30 ... -10	
X_i	kg/kg	0.012 ... 0.001	
Cool air outlet			
p	bar	1.0 ... 0.5	
Return air inlet			
\dot{m}	kg/s	1.0	
ϑ	°C	24	
X_i	kg/kg	0.008	
Chilled air outlet			
\dot{m}	kg/s	2.0	

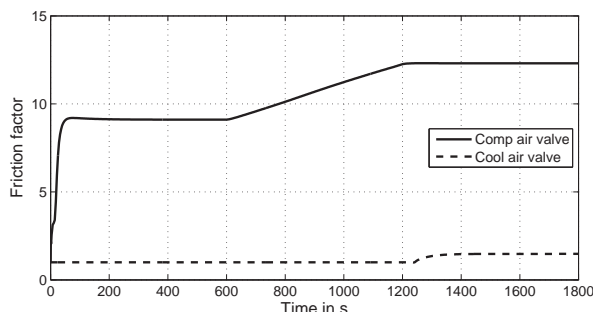


Figure 8: Friction factors of valves due to decrease of relative opening

,straight pipe, etc.) fulfill the requirements of being used in large thermo-hydraulic systems like environmental control system.

5.3 Summary

The concept of the a new Modelica library called FluidDissipation has been presented. FluidDissipation is developed in the European research project

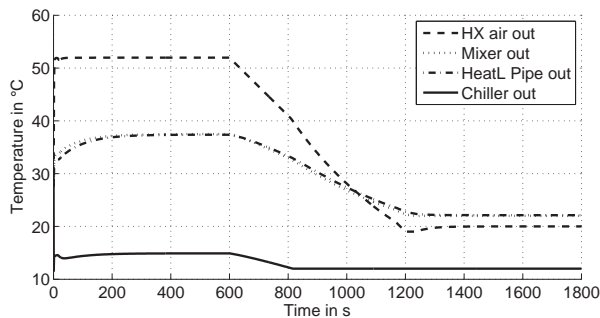


Figure 9: Transient temperatures of compressed air from HX outlet to chiller outlet

Eurosyslib-D and will be a free library for calculation of heat transfer and pressure losses of energy systems. First examples like fluid flow in straight pipes and a simple environmental control system have shown possible applications for energy systems. Further tasks are the enhancement of more heat transfer and pressure loss calculations of energy devices. For the verification and validation measurement results have to be supplied.

The project underlying this report is funded by the German Federal Ministry of Education and Research under the grant no.[IS07003C]. The authors bear the sole responsibility for the content of this publication.



References

- [1] St. Wischhusen. *Dynamische Simulation zur wirtschaftlichen Bewertung von komplexen Energiesystemen*. PhD thesis, Technische Universität Hamburg-Harburg, AB Technische Thermodynamik, Hamburg, 2005.
- [2] H. Elmquist, M. Otter, and S. E. Cellier. In-line integration: A new mixed symbolic/numeric

approach for solving differential-algebraic equation systems. In *Proceedings of European Simulation MultiConference*, Prague, 1995. ESM'95, ESM'95.

- [3] I. E. Idelchik. *Handbook of hydraulic resistance*. Jaico Publishing House, Dehli (India), 3. edition, 2006.

Session 2c

Mechanical Systems & Applications

Development of an Aircraft and Landing Gears Model with Steering System in Modelica-Dymola

Gianluca Verzichelli
Airbus UK Ltd.
BS99 7AR, Filton. United Kingdom
gianluca.verzichelli@airbus.com

Abstract

This paper describes one of the first uses of Modelica, with Dymola, for modelling and simulation activities of landing gears in Airbus. The application of Dymola was for the development of a model of the whole Aircraft and the auxiliary and main landing gears, including tires, wheels, oleo-pneumatic shock absorbers, airframe, etc.

The suitability of Modelica for describing model at system level has been exploited. In this case, it has provided steering functions for the whole Aircraft, with the development of the Nose and Body Wheel Hydraulics Steering System connected to the mechanical domain. Furthermore, most of the electrical components, part of the Control and Monitoring System of the Aircraft, have been taking into account, so that the interaction between the electrical, hydraulics and mechanical domains forms a close link using one modelling language. The model has been developed using mainly the free library *Mechanics* and the commercial library *HyLib*.

Keywords: *aircraft; landing gear; steering system; simulation; modelling.*

1 Introduction

Over the last three decades, civil aircraft systems have become progressively more integrated, encompassing several different domains: structure, power, control and software. In a such tightly coupled environment the use of one modeling and simulation language like Modelica, can provide tangible advantages for the engineers. In particular, it can decrease the lead-time to develop several *ready-to-use* architectures of the Aircraft model, with different levels of detail, so that the engineers can investigate many more *what-if* strategies with high level of accuracy of the analysis, thus minimizing the risks of the *design-build-test-fix* cycle,

which is an expensive, uncompetitive, unpredictable – and ultimately prone to failures, paradigm of product development.

2 Aircraft Library

The following sections highlight the main features of the sub-models developed for the Aircraft Library. Any of them can be placed in a super-model to create a detailed architecture of a desired aircraft which then can be tested under the foreseen simulated operative conditions, see Figure 1.

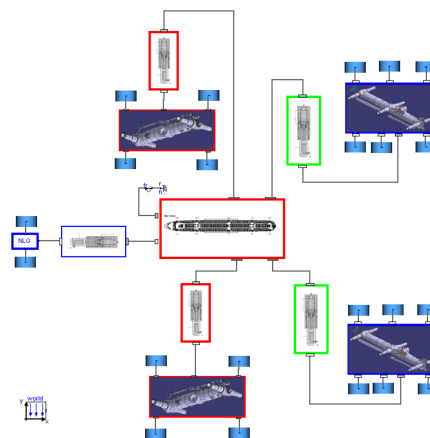


Figure 1: Snapshot of the full model in Dymola

2.1 Airframe Model

The Airframe Model contains the mass, inertia tensor and the main geometrical characteristics of the Aircraft (A/C) like wheelbase, track, etc. The mass is lumped and concentrated at the Centre of Gravity (CG) location and the inertia tensor is calculated with respect to the CG. The user can define the position of the CG (yellow sphere) with respect to the Mean Aerodynamic Chord (MAC) of the A/C (purple segment),

typically between 35% and 42% of the MAC, see Figure 2.

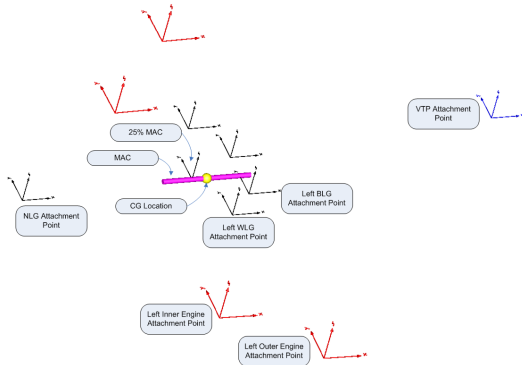


Figure 2: Airframe Model in Dymola with parameterized attachment points

In addition the user can define the attachment points of the Nose, Wing, Body Landing Gears, the Engines and the Vertical Tail Plane Pressure Centre with respect to the Aircraft Datum, see Figure 3.

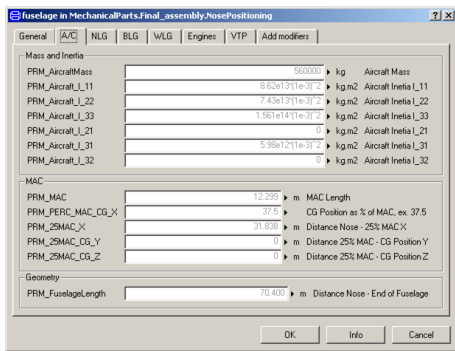


Figure 3: GUI for definition of the Airframe Model main parameters

The approach to lump the whole mass and inertia tensor of the A/C at the CG location is quite common and straightforward, it is also typical of the models built with a top-down approach. This method implies an extensive use of the *FixedTranslation* Block, with no mass and inertia: the only mass and inertia are in the Airframe Model. Though this is theoretically correct, Modelica could experience problems solving the equation of motions in all the cases where there is a *local* translation or rotation of two parts: for instance, the rotation of the steerable aft axle of the Body Landing Gear with respect to the Bogie. This method is also unadvisable in all those cases when the models are built with a bottom-up approach (many sub-models which will be used to develop a top level model): in these cases, each part of the sub-models should have its own correct, or at least realistic, mass and inertia, so that

the sub-model can be verified and validated in isolation.

2.2 Shock Absorber Model

The Shock Absorber is represented by means of an oleo-pneumatic suspension model. Its characteristics vary with relative displacement, velocity and direction of travel of the sliding cylinder with respect to the outer cylinder. The model contains characteristics for stiffness and damping for each of the landing gears as supplied by the vendors (polytropic dynamic curves). Stiffness force is calculated as a function of oleo extension displacement via evaluation of spring stiffness curves, see Figure 4.

Damping force is calculated as a function of oleo extension displacement, the rate of change of oleo extension displacement and oleo extension displacement direction. To achieve this, two damping coefficient curves are used, one defining the compression stroke and one defining rebound stroke, see Figure 5 and Figure 6. Oleo damping force is then calculated by multiplying this coefficient by velocity squared.

At the top-level, the user can define the rake angle of the shock absorber and other key characteristics (maximum stroke, sliding cylinder length, etc.).

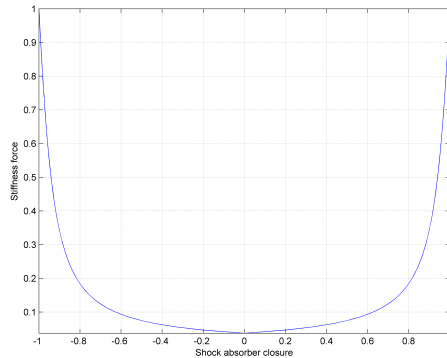


Figure 4: Stiffness force vs. shock absorber closure at 20° Celsius (Normalized)

2.3 Bogie Model

The main functionality of the Bogie model is to provide attachment points for the wheels and the bottom of the shock absorber so to create a correct load path distribution of the weight of the A/C on the ground. The user can choose between a Dual, a Dual Tandem or a Tri-Twin Tandem Bogie. Obviously other types of

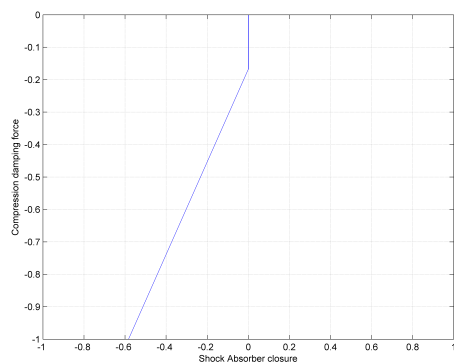


Figure 5: Compression damping force vs. shock absorber closure at 20° Celsius (Normalized)

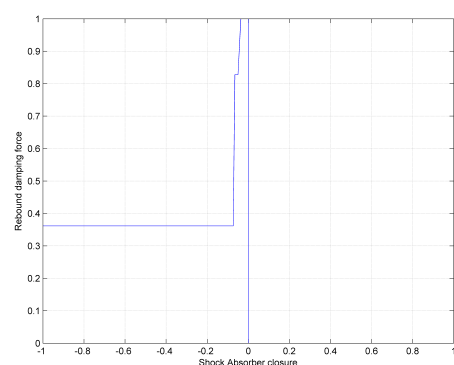


Figure 6: Rebound damping force vs. shock absorber closure at 20° Celsius (Normalized)

Landing Gears Wheel Layouts model can be easily developed: Dual Twin, Dual Twin Tandem, etc. In the case of the Body Wheel Steering (BWS), the aft axle is steerable, see Figure 7, so the model provides extra functionalities: these are explained in § 2.3.1.

The user can define the main geometrical characteristics like track and wheelbase and whether consider it as massless and with zero inertia or not.

2.3.1 Body Landing Gear Model

The Body Landing Gear (BLG) Model has extra functionality because of its characteristic of having the aft axle steerable. It includes the *LineForceWithTwoMasses* Block for the inclusion of the steering actuator and lock actuator characteristics (lumped masses of cylinder and piston) and connection to the hydraulics actuator of the Steering System Model, as well as Return springs. In addition, a *brake* Block has been used to simulate the status of locked condition of the aft axle. When the Avionic System commands to lock the aft axle, supposed to be initially unlocked, a command

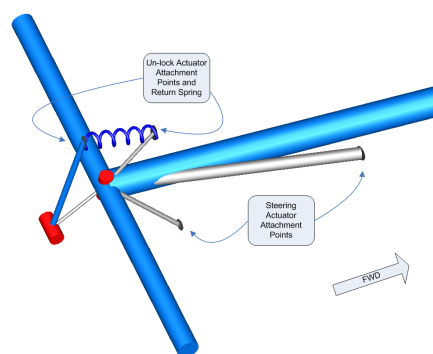


Figure 7: Tri-Twin Tandem Bogie Model with steerable aft axle and lock system

to retract the lock actuator is sent.

The Lock system uses a wedge beam that pivots a one end attached to the bogie whilst the other end is attached to the locking actuator, see Figures 7 and 8.

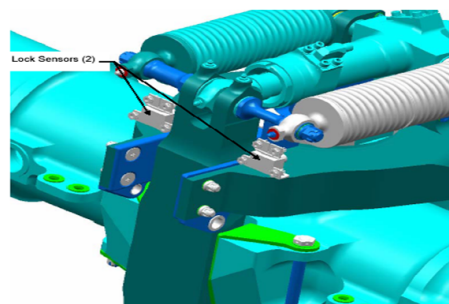


Figure 8: Lock System

As the lock beam is drawn towards the aft axle via the Return springs (retraction of the lock actuator) it engages its wedge into a V-shape aperture in the axle causing it to be locked. As the lock beam is deployed away from the aft axle (extension of the lock actuator) the wedge becomes free from the aperture in the axle allowing the axle to rotate freely. The return springs not only reduce the closing force needed to lock the locking wedge but also maintain the locking wedge in its closed position.

At the moment, given a certain part, Modelica does not recognize the surface of it but only *frame a* and *frame b*, together with the position of the CG with respect to *frame a*, the mass and inertia tensor of the part. So the simulation of surfaces contact/collision is not possible unless the point, or points, of contact/collision remain known during the whole simulation.

2.4 Tyre Model

The modelling and simulation activity of a tyre, and in particular of an aircraft tyre – subject to higher slip

angles with respect to automotive tyres – represents a difficult task. Many different tyre models have been developed in Modelica: Magic Formula model, Rill Model, Brush Model, see [1], with different level of accuracy and computational effort. The model developed for the Aircraft Library represents a good compromise between accuracy and simulation time and it seems to be suitable for on ground manoeuvrability studies. Obviously, thanks to the modularity development feature of the Modelica language, when a more detailed model of the tyre will be developed, this can be easily and quickly implemented in the full A/C model.

The model developed makes use mainly of look up tables with empirical data extracted from tyre suppliers manufacturers. The model calculates Lateral Slip Angle, Side and Drag Forces, Vertical Reaction and Self Aligning Torque. The point of application of the Side, Drag Forces and Self Aligning Torque is fixed and it is coincident with the vertical projection of the axle hub at a vertical distance equal to deformed radius of the tyre. The runway and taxiway are assumed to be flat and the camber angle is neglected. The model allows the user to chose the value of the coefficient of rolling resistance, the tyre deformed radius, the inflation pressure, the tyre damping coefficient, the mass and inertia of the tyre.

2.4.1 Lateral Slip Angle Computation

The Lateral Slip Angle is the angle formed by the plane of rotation of the wheel and the tangent to the wheel's path, see Figure 9. Mathematically can be expressed as:

$$\alpha_y = \arctan(V_y/V_x) \quad (1)$$

where V_y and V_x are the velocities of the footprint of the resultant of the pressure distribution forces between tyre and ground. The model uses the simplified assumption explained in § 2.4, so it assumes that there is no elastic deformation of the tyre belt for the computation of the lateral slip angle.

The value of (1), computed by the model, is different from zero only when there is contact between tyre and ground and the lateral and longitudinal velocities of the tyre footprint are above a certain threshold. In addition, the model assumes an instantaneous development of steady state reaction forces and momentum.

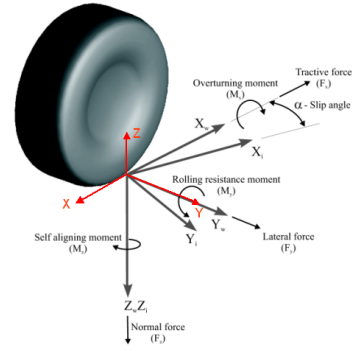


Figure 9: Tyre axis system as defined by the SAE

2.4.2 Vertical Reaction Computation

The Vertical Reaction is calculated by means of a non-linear spring behavior using the following equation:

$$\text{Max}(0, k \cdot (q - q_0)^e - c \cdot \dot{q}) \quad (2)$$

where k is the stiffness coefficient, c is the damping coefficient, e is a coefficient depending on the type of material, q is the actual displacement variable, q_0 is the tyre deformed radius, \dot{q} is the actual velocity variable. Another model computes the dynamic Vertical Reaction of the tyre using (2) but substituting the stiffness behavior $k \cdot (q - q_0)^e$, with the look-up table of Figure 10, which takes into account the inflation pressure and the actual deflection of the tyre.

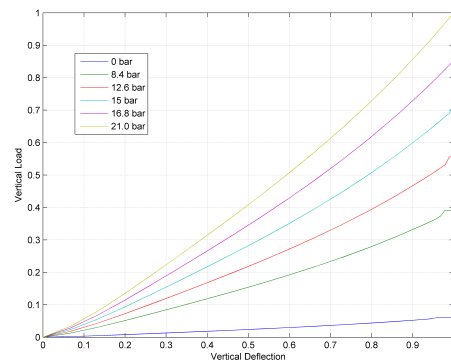


Figure 10: Dynamic Vertical Load vs. Vertical Deflection for different Inflation Pressures (Normalized)

The user can choose between the two different methods of computation because the model has been made *replaceable*.

The Vertical Reaction sustained by each tyre and the lateral slip angle are used to calculate the tyre forces and torques as explained in § 2.4.3.

2.4.3 Side, Drag Forces and Self Aligning Torque Computation

The equations which compute the drag and side forces are as follow:

$$D = \mu_R \cdot F_z + D' \cos(\alpha_y) - S' \sin(\alpha_y) \quad (3)$$

$$S = S' \cos(\alpha_y) + D' \sin(\alpha_y) \quad (4)$$

where μ_R is the rolling friction coefficient, F_z is the Vertical Load, D' and S' are the drag and side forces resolved in the wheel plane reference system and α_y is the slip angle. Notably, there is no need to manipulate them if α_y changes sign, because of the trigonometric functions \sin and \cos and the shape of the curves of Figures 11 and 12. Differently, if the A/C changes the direction of motion, i.e. instead of moving forward, moves backward, during *push-back* manoeuvres for instance, there is the need to change the sign of the previous equations. For this reason, the RHS of (3) and (4) is multiplied by $(-\tanh(\tau \cdot V_x))$, where τ is a time constant chosen by the user and the \tanh is used to assure a smooth transition around zero (the A/C is assumed to move forward when $V_x < 0$). The final equations are:

$$D = [\mu \cdot F_z + D' \cos(\alpha_y) - S' \sin(\alpha_y)] \cdot [-\tanh(\tau \cdot V_x)]$$

$$S = [S' \cos(\alpha_y) + D' \sin(\alpha_y)] \cdot [-\tanh(\tau \cdot V_x)]$$

The torque developed by the Self Aligning Torque can be simply computed using the look-up table of Figure 13. A snapshot of the tyre model in Dymola is given in Figure 14.

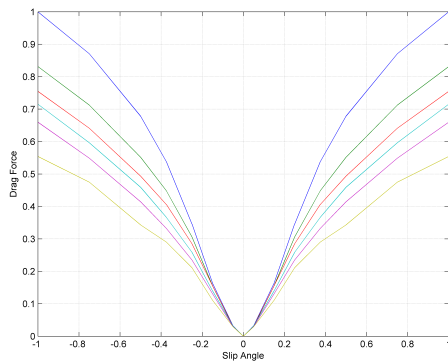


Figure 11: Drag Force vs. Slip Angle for different Vertical Load (Normalized)

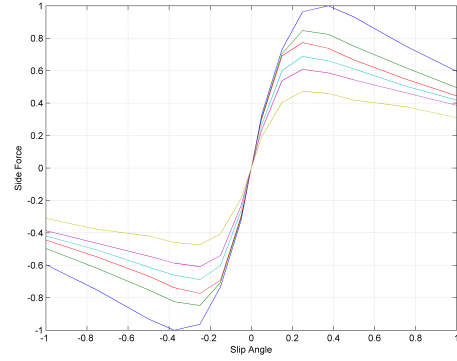


Figure 12: Side Force vs. Slip Angle for different Vertical Load (Normalized)

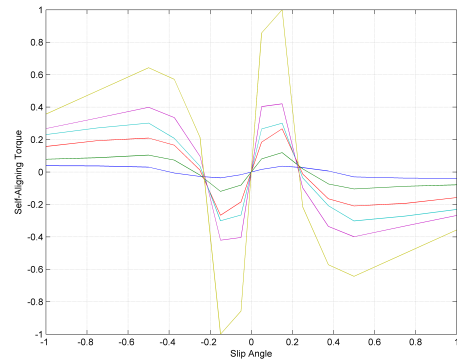


Figure 13: Self Aligning Torque vs. Slip Angle for different Vertical Load (Normalized)

2.5 Engine Model

The Engine Model is simply represented by a PID Controller. The user can define the desired steady state velocity and the spool time: the engine forces change accordingly to maintain the value of the demanded velocity. In fact during a turn, the A/C speed tends to decay because of the development of the centripetal force at the tyre contact footprint which counterbalances the centrifugal force. The user can also decide when to switch the engine on ($Thrust \neq 0$) or off ($Thrust = 0$) due to conditions linked either to time-based events or boolean-based events. Furthermore, in order to simulate an instantaneous loss of thrust, due for instance to an engine failure for the simulation of a rejected take-off case, the *TriggeredMax* block is used. It samples the continuous input signal whenever the trigger input signal is rising (i.e., trigger changes from false to true). The maximum, absolute value of the input signal (Engine Thrust) at the sampling point is provided as output signal. So the Thrust on the remaining functioning engines is assumed to be constant and its value equal

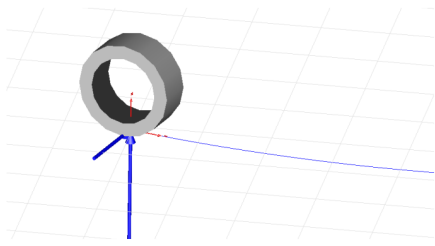


Figure 14: Tyre Model in Dymola

to the one at the instant when the failure occurred.

2.6 Aerodynamic Model

The Aerodynamic forces and momenta are lumped at the CG location. The equations used are as follows:

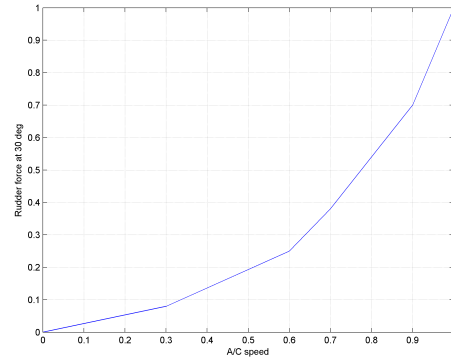
$$\begin{aligned}
 F_{xA} &= \frac{1}{2} \rho V_{CG}^2 S C_x \\
 F_{yA} &= \frac{1}{2} \rho V_{CG}^2 S C_y \\
 F_{zA} &= \frac{1}{2} \rho V_{CG}^2 S C_z \\
 M_{xA} &= \frac{1}{2} \rho V_{CG}^2 S b C_l \\
 M_{yA} &= \frac{1}{2} \rho V_{CG}^2 S \bar{c} C_m \\
 M_{zA} &= \frac{1}{2} \rho V_{CG}^2 S b C_n
 \end{aligned} \tag{5}$$

where S is the wing wet area, b the wing span, and \bar{c} the mean aerodynamic chord. These equations are multiplied with a positive or negative sign to account for the axes reference system. The computation of some of the angles necessary for the evaluation of the coefficients of (5) is as follow:

$$\alpha = \arctan\left(\frac{u}{w}\right) \quad \beta = \arcsin\left(\frac{v}{|V_{CG}|}\right)$$

Other angles and their rate of change are computed using a similar approach.

The rotation of the Rudder is taken into account separately using the look-up table of Figure 15. It is assumed that the rudder has full authority when its rotation reaches 30° , and it increases linearly from zero to 30° . The rotation of the rudder implies also a rotation of the nose wheel, this has been implemented in the Steering Laws in the Monitoring and Commanding Model.

Figure 15: Rudder Force at 30° Rotation vs. A/C Speed (Normalized)

2.7 Hydraulics Steering System Model

The Hydraulics Steering System Model consists of a Nose Wheel Steering (NWS) and a BWS system. For the first one the kinematics of the actuation consists of a push-pull actuator arrangement which is capable to steer the A/C from a straight ahead position ($\theta_{NWS} = 0 \text{ deg}$) to a full powered steering rotation of the Nose Wheel ($\theta_{NWS} = \pm \theta_{NWS_{max}}$). For the second one, the kinematics consist of a single linear actuator, which steers the BWS accordingly to the actual position of the NWS angle and ground speed of the A/C ($\theta_{BWS} = f(\theta_{NWS}, GS_{A/C})$). An overview of the NWS and BWS Steering System models is given hereafter.

2.7.1 NWS Hydraulics Steering System Model

At the top-level, the system briefly consists of a Normal Selector Valve Manifold (NSELVM), an Alternate Selector Valve Manifold (ASELVM), a Local Electrohydraulic Generation System (LHEGS), Nose Landing Gear (NLG) Shutoff-Swivel Valve, an Hydraulic Control Block (HCB), a servo-valve electro-hydraulic NWS, two Change Over Valves and two steering actuators. In addition, the system is connected to the hydraulic power distribution system via the High Pressure ($\approx 350 \text{ bar}$) and Low Pressure Manifolds ($\approx 5 \text{ bar}$). All the electrical signals necessary to energize and de-energize the various selector valves and command the servo-valve are sent by/to COMMON to the Hydraulics System. In turn, the latter transmits all the signal necessary to monitor the system to COMMON, for instance the value of the pressure downstream NSELVM, via Pressure Transducer (PT) PT4.

In the model, all the previous elements have been modeled, except the ASELVM and the LHEGS, which are

mainly necessary only if particular faults in the system occur (Reversion from Normal to Alternate Mode) and they will be modeled in the future, see Figure 16.

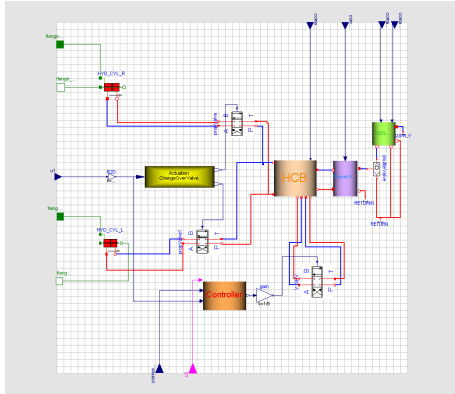


Figure 16: NWS Hydraulics Steering System Model in Dymola

The model of COM/MON does not pretend to be exhaustive nor representative of the whole COM/MON system, its principal task is to support the functioning of the Hydraulics System Model. The Model has been built keeping a net interface between the hydraulic domain, the mechanical domain and the avionic domain: the green flanges represent the interface with the attachment points of the mechanical steering actuators and the blue and purple signals the interface with the avionic domain, see Figure 16.

The position of the spool of the servo-valve is controlled by a PI controller which modulates the current in order to minimize the error between NWS commanded angle ($\theta_{NWS_{com}}$) and actual NWS angle (θ_{NWS}). The position of the servo-valve spool allows the hydraulic flow to differentially pressurize the four chambers (Left and Right, Annulus and Full Bore) of the steering actuators, so to create push or pull forces on the pistons ends.

The attachment points of the actuators are fixed: to the stationary flange for the cylinders and to the rotating sleeve for the pistons, respectively. This, and their position with respect to the upper strut of the shock absorber, are such to create a steering torque, which is transmitted, via the torque links, to the bottom of the shock absorber strut (piston fork) allowing the A/C to steer, see Figures 17, 18 and 19.

Of particular interest is the situation when one of the two actuators stalls: the line of action of the hydraulic force intersects the axle of rotation of the strut, creating no moment arm. The angle at which this happens is called *Change Over Angle* (θ_{COV}), and it is equal to:

$$\theta_{COV} = \theta_0 - \arctan(Fr_y/Fr_x)$$

with θ_0 being the angle between the x axis of the nose landing gear reference frame and the vector $S_r O_N$, Fr_x and Fr_y equal to the x and y coordinates of Fr . These coordinates given in a reference system with the $x - y$ plane orthogonal to the strut axle (if the rake angle is different from zero).

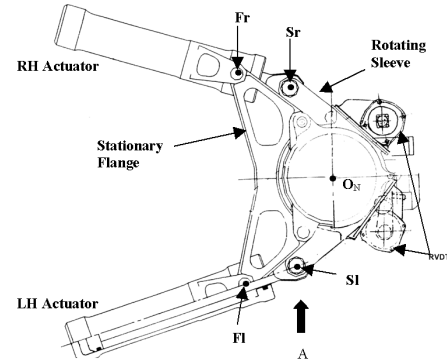


Figure 17: Actuators-rotary sleeve assembly

When $\theta_{NWS} = \pm\theta_{COV}$, one of the two Change Over Valve receives the command to change its position, see Figure 16, either the left or the right one depending if the A/C is performing a clockwise turn or counter-clockwise turn, so that the corresponding actuator begins to push or pull and viceversa.

2.7.2 BWS Hydraulics Steering System Model

At the top-level, the system briefly consists of a Steering Selector Valve Manifold (SSELMV), a left and right BWS Hydraulic Control Block including Selector Valves and Steering Actuator, a left and right lock actuator, a left and right BWS electro-hydraulic servo-valve, an High Pressure supply line (HP) and Low Pressure supply line (LP) Manifolds and ATA 29 Electro-Motor Pump (EMP), see Figure 20.

Whilst the steering functions of the BWS is similar to the NWS, except the fact that there is one linear steering actuator per side, the additional challenge has been in modelling the lock/un-lock mechanism and un-lock actuator, as already explained in § 2.3.1, see Figure 8.

2.8 Monitoring and Commanding Model

The Monitoring and Commanding Model developed at this stage has the main purpose of allowing the correct functioning of the hydraulic and mechanical Model.

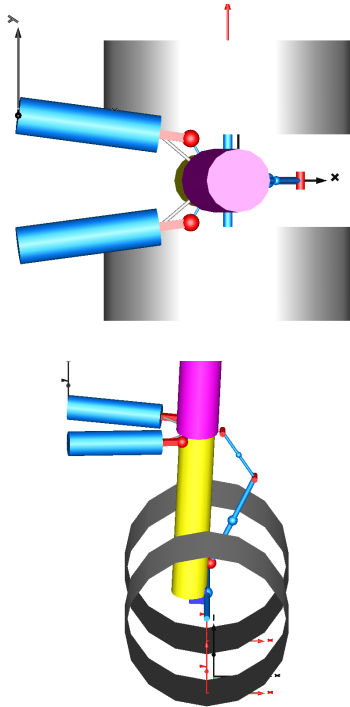


Figure 18: NLG Model with Actuators-rotary sleeve assembly in Dymola

The signals to energize or de-energize the several selector valves are sent accordingly to the actual conditions of the A/C, so to create a closed loop between the main three domains. The implementation of the Steering Laws has been also developed in this model, see for instance the command of the BWS angle in accordance with the actual NWS angle, Figure 21.

A more comprehensive model will be developed in a second stage: in doing that, an extensive use of the *StaeGraph* Library will be pursued.

3 Simulation Results

The following sections highlight the Dymola set up used to run simulations and the results obtained from a simulation of a Rejected Take Off (RTO) scenario.

3.1 Simulation Settings

The type of integration algorithm used depended upon the model that was tested. In the case of pure mechanical model, the *Dassl* or the *Lsodar* algorithms performed in an excellent manner. When it came to simulate the full model, combining avionic, hydraulic and mechanical domains, the best performance has been achieved using the *Sdirk34hw* method. Notably, the increase of tolerance of the integrator did not improve

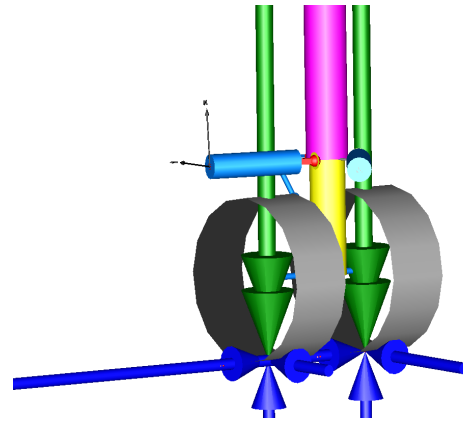


Figure 19: NLG driven by hydraulics actuators at 70° with side, drag and vertical forces (blue) and self-aligning torque (green) in Dymola

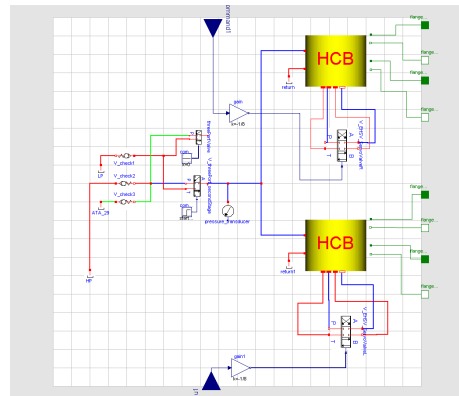


Figure 20: BWS Hydraulics Steering System Model in Dymola

the simulation time: as explained in [2], a condition of optimum should exist, in the case of the full model a tolerance of 10^{-5} was used. Quite challenging has been the identification of the right initial conditions: Dymola by default, assigns arbitrary values for the initial conditions of certain variables. The assumption made by the software should always be validated by the user. In the case of the full A/C model, at the beginning of the simulation, there are many events that occur: the A/C is settling down on the soil (impact force different from zero and the shock absorber starts to be compressed), the A/C speed begins to reach the steady state value, and especially, the hydraulic circuit tends to find the steady state condition. All this could be quite time consuming from a simulation point of view. Ideally the user should try to find the value of the variables in the steady state condition which would like to use as starting point of his/her investigation, record those values, and use them as initial conditions for all the following simulations. When this is done, he/she

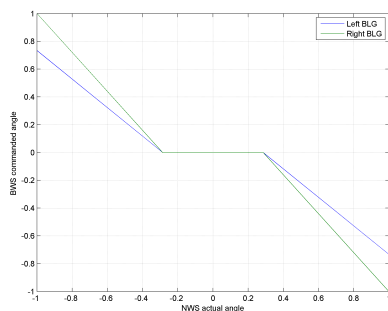


Figure 21: BWS Steering Laws

could simulate the model up to the steady state condition and then used this final condition, as a starting point for a new simulation. This method can be used only if the model has reached an high level of maturity and many changes are no longer necessary.

3.2 RTO simulation test

The test has been carried out with the following pre-conditions and assumptions: A/C speed at the instant of the left outer engine failure equal to 130 kt , mass equal to 560 t , CG position equal to 37.5% , rolling friction coefficient μ_R , equal to $6 e^{-3}$, all the steering controllers active, ($\theta_{NWS_{com}} = 0$) with no pilots correction after engine failure, no aerodynamic forces and rudder force.

The main outputs of interest are the trajectory of the nose and CG, Figures 23 and 24, A/C heading ψ , Figure 27; A/C CG lateral linear and angular accelerations, a_{yCG} and $\dot{\omega}_{zCG}$ respectively, Figures 25 and 26. Notably, It takes 1.25 s before the CG linear acceleration along y direction starts to become negative, see Figure 25, in fact the CG first moves towards the positive $x-z$ half plane and then after a while, it starts to move towards the negative $x-z$ half plane, following the nose. After three seconds from the instant of the simulated engine failure, the nose has moved of circa 37.22 cm towards the left.

4 Future Work

The future work will be mainly based on the enhancement of the Aircraft Library with the development of more comprehensive models and new models as well; on the investigation of Modelica capability to produce code to run models in a Real Time environment for hybrid simulations on the landing gear test rig and, finally, on the validation of the models with Flight Test

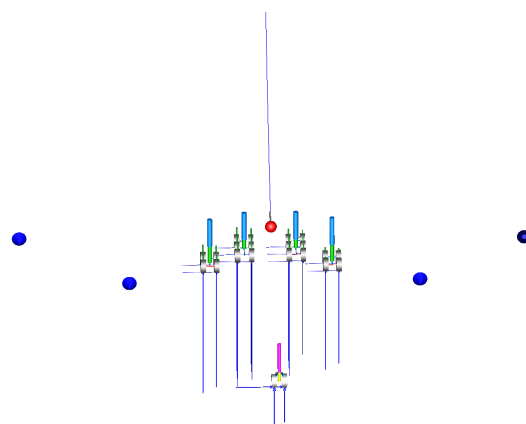
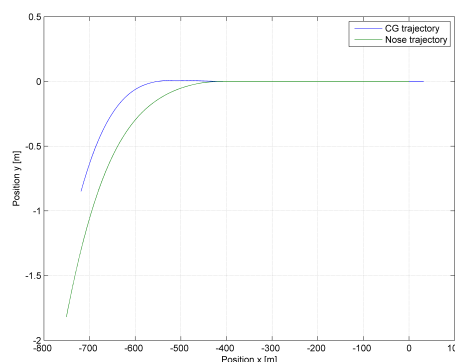


Figure 22: Snapshot of the A/C, five seconds after a left outer engine failure with highlighted CG trajectory (in blue)

Figure 23: CG and Nose trajectory (engine failure at $x = -382.47 \text{ m}$. A/C moves forward when $x < 0$)

and Test Rig Data.

5 Conclusions

Though the work presented in this paper is one of the first large-scale application of Modelica for the simulation of landing gears and aircrafts, its findings made clear the power and potentiality of the language to model and simulate so tightly coupled systems such the ones which equip new modern airplanes.

The fact that the source code is completely open to the user implies a huge potentiality to increase the level of accuracy of every model component/assembly/system. The acasuality feature of the language implies a real opportunity to develop a re-usable library of models which is not truly possible with models built with a casual language. Furthermore, the system modeled with Modelica provides a unique feature which is the capability of the model to *look like* the real system undergoing the design process: in a large enterprise this

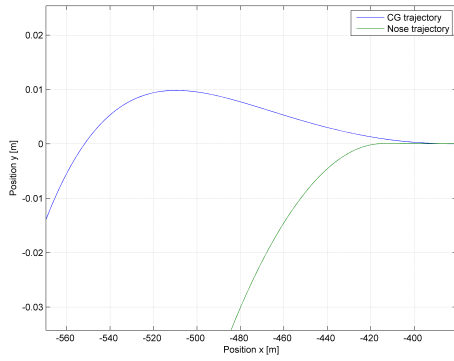


Figure 24: CG and Nose trajectory (engine failure at $x = -382.47 m$. A/C moves forward when $x < 0$) (Magnified)

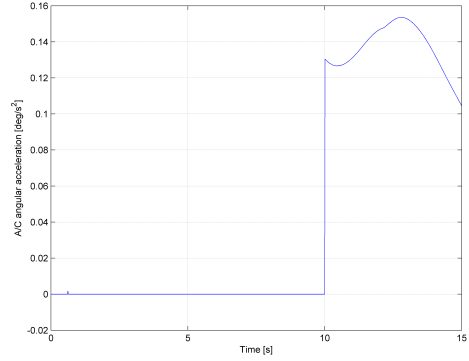


Figure 26: A/C angular acceleration along z at CG location

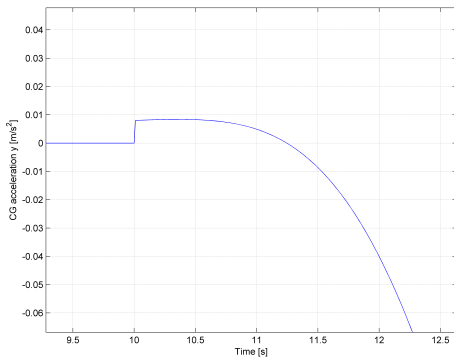


Figure 25: A/C linear acceleration along y at CG location (Magnified)

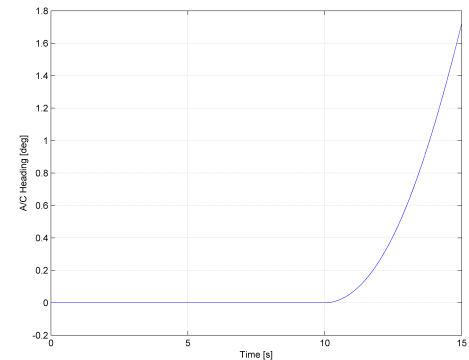


Figure 27: A/C heading angle ψ

represents a *point of convergence* between designers and modelers. No particular skills are necessary to be able to *read* the model which faithfully should represent the system, so that feedbacks, critics, enhancements, approvals can be easily agreed between several actors not necessarily experts of modelling and simulations techniques, even before any type of simulation is performed.

If Modelica is used in a large industrial environment some enhancements are advisable, though these issues may not be necessarily attributable to Modelica itself: the capability of creating geometry (with mass, inertia, CG location and surface shape information) directly from a Computer-Aided Drawing (CAD) software in an automated manner; a flexible body simulation capability, including contacts and conditional connection/dis-connection of bodies/joints during the same simulation.

6 Acknowledgments

The Author would like to acknowledge Airbus for giving the opportunity to attend the Modelica 2008 Conference and present this work, Mr. Sanjiv Sharma for the continuous support and for reviewing the paper, Miss. Serena Simoni for having accepted the challenge of learning Modelica and Dymola and started this work and finally Claytex Services Limited and Dynasym AB for solving technical issues related to the use of the software.

Acronyms

- A/C Aircraft
- ASELVM Alternate Selector Valve Manifold
- BLG Body Landing Gear
- BWS Body Wheel Steering
- CAD Computer-Aided Drawing

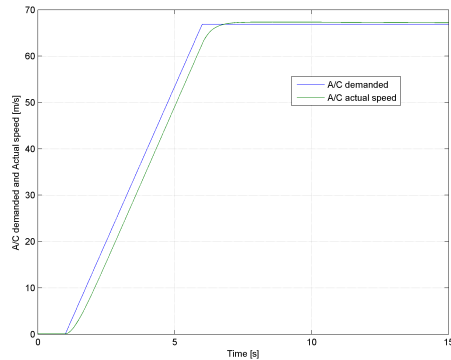
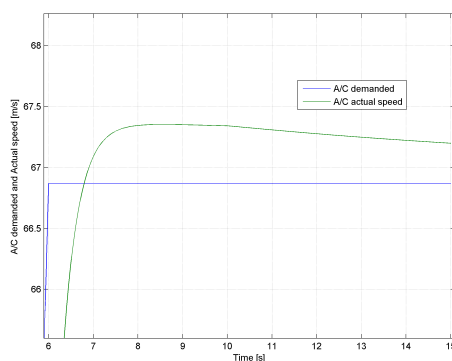


Figure 28: A/C demanded and actual speed [m/s]

Figure 29: A/C demanded and actual speed [m/s]
(Magnified)**CG** Centre of Gravity**EMP** ATA 29 Electro-Motor Pump**GUI** Graphical User Interface**HCB** Hydraulic Control Block**HP** High Pressure supply line**LHEGS** Local Electro-hydraulic Generation System**LP** Low Pressure supply line**MAC** Mean Aerodynamic Chord**NLG** Nose Landing Gear**NSELVM** Normal Selector Valve Manifold**NWS** Nose Wheel Steering**PT** Pressure Transducer**RTO** Rejected Take Off

References

- [1] M. Beckman and J. Andreasson. Wheel model library for use in vehicle dynamics studies. KTH Vehicle Dynamics, Sweden
mb.johang@fkt.kth.se
- [2] C. Clauß and P. Beater. Electronic, Hydraulic, and Mechanical Subsystems of a Universal Testing Machine Modeled with Modelica. 2nd International Modelica Conference, Proceedings, pp. 25-30, Germany

The New DLR Flight Dynamics Library

Gertjan Looye

German Aerospace Center (DLR), Institute of Robotics and Mechatronics
Oberpfaffenhofen, 82234 Wessling, Germany

Abstract

An overview of the new Modelica Flight Dynamics Library of the German Aerospace Center DLR is given. This library is intended for construction of multi-disciplinary flight dynamics models of rigid and flexible flight vehicles. The environment models provide the functionality to cover on-ground operations up to flight at high speeds and high altitudes. The resulting models may be used in various fields and stages of the aircraft development process, like flight control law design, as well as for real-time flight simulation.

Keywords: flight dynamics; flight control; simulation; aeroelasticity

1 Introduction

Dynamic simulation plays an important role in the aircraft design and certification process. Typical examples are development of flight control laws, flight loads analysis, specification and testing of on-board systems, aircraft handling qualities and system assessment in real-time manned flight simulators. DLR has developed an extensive Modelica library that allows for construction of suitable aircraft flight dynamics models for the various stages and applications in the aircraft development process. The following strengths of Modelica have hereby been exploited particularly:

- *Multi-disciplinary modelling:* multi-disciplinary interactions play an important role in flight dynamics. Especially when flight control laws are involved, aspects like flight mechanics, structural dynamics, and systems may show considerable dynamic interactions that must be appropriately addressed in the model used for design analyses. Modelica provides an ideal basis to develop such models, since a large amount of discipline-specific libraries is available that may be used to construct components of the integrated model;

- *Single source modelling:* Especially in flight controls various types of analysis methods are used that require aircraft models to be available in various forms. Examples are nonlinear models for simulation, (symbolic) linear models for stability and robustness analysis, inverse models for control law synthesis, etc. Various forms of models often involve independent implementations of the same model data. Modelica allows for the construction of a single model from which appropriate analysis models may be generated with the help of a model translator.

Back in 1995 the DLR institute of Robotics and Mechatronics developed a first library (at the time, based on the Dymola language) for modelling of aircraft flight dynamics [26]. Objective was to build a solid basis for constructing integrated dynamic aircraft models, including flight dynamics, detailed on board system dynamics, structural dynamics, etc. First applications were a generic transport and a fighter aircraft [20] and a first flexible aircraft shortly thereafter [15]. Since then, the library has been expanded and applied to complex aircraft models that include e.g. system hydraulics and electronics [25, 27, 24]. In the frame of the international projects REAL (Robust and Efficient Autopilot control Laws design, funded by the EU in the fifth frame work programme [31]) and the GARTEUR action group AG-11 on clearance of flight control laws [11], the automatic generation of inverse models for fast trim computation and nonlinear control laws was applied for the first time. Recent application examples are the thrust-vectoring X-31A high-angle of attack experimental aircraft and a real-time capable integrated flight dynamics and aeroelastic transport aircraft model, including unsteady aerodynamics, structural dynamics, control system, etc.

The latest version of the Flight Dynamics Library includes various major enhancements as compared with previous versions. Firstly, it exploits more recent Modelica features, like the bus concept and the matured inner-outer concept. Secondly, the library is

now compatible with the Multibody Library (part of the Modelica standard library), allowing for easy construction of airframes. Thirdly, its *world* model has been enhanced to comply with inertial standards like WGS84 [4] for position reference.

This paper discusses these latest developments, new features, and example applications of the Flight Dynamics Library. In the following section the selection of a generic aircraft model structure is discussed (Section 2), based on which the Flight Dynamics Library has been organised (Section 3). In Section 4 the automatic generation of model code for model simulation and analysis is discussed, followed by some example applications. Finally, a summary is given in Section 6.

2 The aircraft model structure

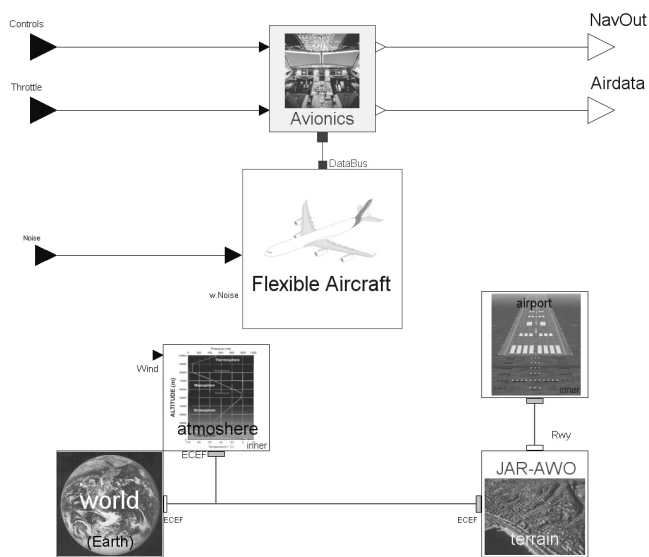


Figure 1: Top-level of model: aircraft and environment

The objective of this section is to introduce and motivate the basic structure of an aircraft model as it may be composed from the Flight Dynamics Library. The structure of this library will be discussed thereafter.

In constructing complex models the choice of hierarchy is crucial, since this largely determines how model components interact. For the Flight Dynamics Library a top-level model structure as shown in Figure 1 has been adopted. It consists of one or more aircraft, and environment objects (The *avionics* component and input and triangular-shaped output connectors will be discussed in Section 4). The environment objects include a *world*, *atmosphere*, *terrain*, and *airport* model. Note that the (in this case, single) aircraft model has no direct link with the environment models, which phys-

ically makes sense. Using the so-called inner-outer feature of the Modelica language, these models provide field functions. For example, the aircraft may request its surrounding atmospheric conditions from the atmosphere model by sending its local inertial position. Any other aircraft (or e.g. sensor) object in the model may do this as well. This is an advantage as compared with most block-oriented libraries, where an atmospheric model is directly linked to, and thus occupied by, the one aircraft. The ability to easily include multiple air vehicles is useful for applications involving mutual interactions, like towed gliders, wake vortices, air-to-air refuelling, release of missiles, etc.

2.1 The world model

In the following subsections the environment models in Figure 1 (*world*, *atmosphere*, *terrain*, *airport*) will be discussed. These components determine validity of the over-all model to a large extent. Most important is the *world* model (in this case the Earth, but the underlying base class may be extended to implement models of other planets), since it provides the inertial reference in the form of the so-called Earth-Centred Inertial (ECI) reference frame. Its origin is attached to the Earth's centre of mass, its orientation is fixed with respect to reference stars. In addition, the model component has the following functions:

- Provide the geodetic reference. As indicated in Figure 1, to this end the World Geodetic System 1984 [4] (WGS84) is used. The object implements an Earth-Centred Earth-Fixed (ECEF) reference frame, which has the same origin as the ECI, but rotates with the Earth. The attitude of the ECEF (w.r.t. ECI) is available in a connector. A set of functions transform ECI and ECEF referenced position vectors into geodetic longitude, latitude, and height co-ordinates (w.r.t. WGS84 ellipsoid) and vice versa. For a given longitude and latitude, another function provides the local undulation of the so-called EGM96 (Earth Gravitational Model 1996) geoid with respect to the WGS84 ellipsoid, providing the Mean Sea Level (MSL) reference [14].
- Implement a model of the Earth's gravitation. The gravitational model to be used with WGS84 is the Earth Gravitational Model 1996 (EGM96), provided in the form of tables describing equipotential surfaces a function of longitude and latitude. Currently, a more simplified height and geocentric latitude-dependent (Ref. [33] -

Eqn.(1.4-16)) and a constant gravity model are available.

- Implement a model of the Earth's magnetic field. This field is required to compute indications of compass models. The model is based on the US National Geo-spatial-Intelligence Agency (NGA) World Magnetic Model (WMM), which is published every five years and predicts the time-varying intensity and direction of the magnetic field as a function of WGS84 longitude, latitude and height. The current model covers 2005 till 2010 [23].

Double-clicking on the *world* object in Figure 1 allows a number of parameters to be set, like whether the Earth is rotating or in rest, initial day time, and the type of gravity model (approximate EGM96, height independent, or constant). The features of the object may be overkill for many applications, but provide sufficient generality for use with for example high speed and high altitude flight vehicles. Furthermore, the applied WGS84 ensures compatibility with standard GPS equipment, with most flight simulator vision systems, navigation system models, etc. Obviously, any parameter set in the *world* and other environment models applies to all components in the aircraft model.

2.2 The atmosphere model

The second environmental object in Figure 1 is the atmosphere. Normally, the International Standard Atmosphere (ISA) as a function of the height above MSL is used. Alternatively, parameters for constant atmospheric conditions may be entered. The air mass is nominally assumed to be in rest with respect to the ECEF, explaining why a connection with the *world* ECEF-connector exists. However, the component also foresees implementation of wind fields. Currently, wind components in northern and eastern directions may be entered at a reference altitude of 100 ft above the Earth surface. A simple Earth boundary layer model logarithmically reduces the wind velocity to zero on the ground.

2.3 The terrain model

To the right in Figure 1 a terrain model has been added. A component containing highly detailed, or simple parametrised models of the Earth's surface may be selected from the library. Depicted in Figure 1 is a terrain model as used for automatic landing control law design and certification, based on EASA CS-AWO spec-

ifications [10]. The location, elevation, direction, and slope of a runway may be specified, as well as slopes and steps in the terrain below the approach path. A simple function call from e.g. an aircraft sensor then returns the corresponding local terrain elevation above MSL or the WGS84 ellipsoid, allowing for computation of for example the reading of the radio altitude sensor.

2.4 The airport infrastructure model

The airport object implements earth-fixed navigational equipment (e.g. VOR, DME, ILS systems at specified locations). In the figure the ILS equipment of the one runway as positioned in the EASA CS-AWO terrain model is included. Specific characteristics like glide slope angle and antenna transmitter positions may be specified via parameters. Any other model object may obtain its local glide slope and localiser deviation via a simple function call.

2.5 Rigid and flexible aircraft models

The core of the model structure is of course the component that represents the actual aircraft. The Flight Dynamics Library foresees the implementation of rigid just as well as flexible ones. A typical model structure for a flexible transport aircraft is shown in Figure 2. The components resemble physical parts an aircraft consists of (airframe, engines, actuators, sensors), and phenomena it is influenced or driven by (kinematics, aerodynamics, wind).

Component interconnections

For mechanical interconnections the connectors from the Multibody standard library [28] are used. For use with flexible airframes an extended version has been implemented that includes generalised co-ordinates and forces. For the engines and sensors the connector represents the point at which the device is attached to the airframe. Aerodynamic forces actually act all over the airframe. However, in flight mechanics usually only the summed effect with respect to some reference point, like the Aerodynamic Centre (AC), is of interest. Also for the aerodynamic forces in the aeroelastic aerodynamic model (to be discussed shortly) in fact only their summed effect is considered due to so-called left-generalisation with rigid and flexible eigenmodes, see Ref. [19]. Therefore, the aerodynamic models need a single connector only.

Kinematics

The backbone of the model depicted in Figure 2 are the

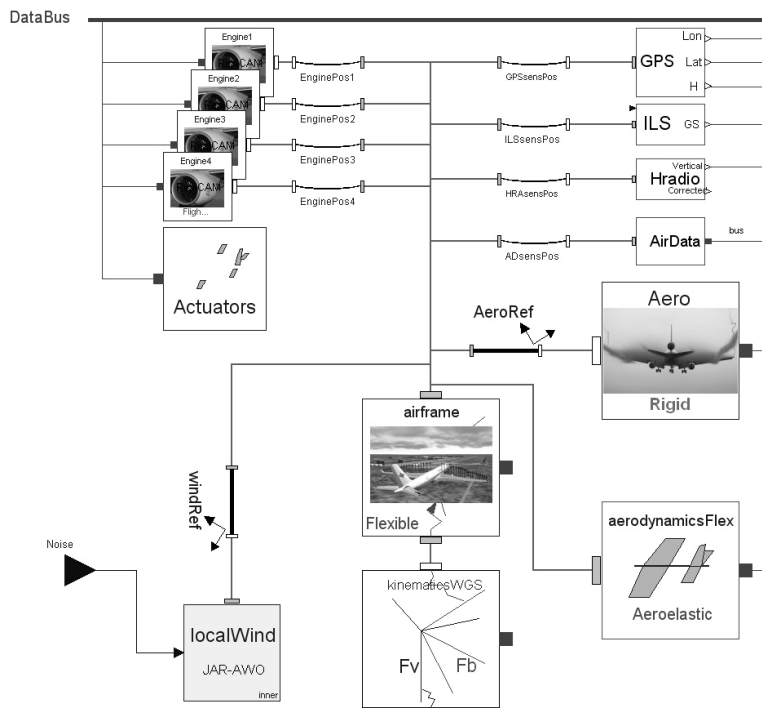


Figure 2: Structure of the *Flexible Aircraft* model in Figure 1

kinematics and *airframe* components. The first defines a “North-East-Down” (NED) local vertical frame with its origin moving with a fixed position in the aircraft, preferably the centre of gravity. The object also defines a right-handed body-fixed reference frame with its origin at the same location, but with a fixed attitude w.r.t. the airframe (x-axis towards the nose, z-axis down). The attitudes and inertial positions of both reference systems are available in the two connectors. The one on top represents the aircraft body reference system, the one below represents the vehicle-carried NED reference frame.

Airframe

The difference between a rigid and a flexible aircraft is, in fact, only in the *airframe* object. In case of a rigid airframe, it contains the standard Newton-Euler force and moment equations with respect to a body reference system [7] (attitude and position in lower connector). Although the origin of this reference system is preferably the centre of gravity (for compatibility with standard flight dynamics models), a fixed point w.r.t. the undeformed airframe shape may be more useful for referencing reasons. The local gravity acceleration is obtained by a call to the *world* object (Figure 1). Note that the computation of gravity depends on the method that is selected in the *world* object. In case of a flexible airframe, linear elastic equations of motion

in modal form augment the Newton-Euler equations [34, 30]. The body axes system is hereby considered as a so-called mean axis system. The momentary shape of the airframe is characterised by states in the form of generalised co-ordinates (also called mode shape multipliers). The underlying data (modal mass, damping, stiffness, and mode shape matrices) are automatically read from a specified file prior to simulation. More details on nonlinear equations of motion of flexible aircraft can be found in Refs. [19, 30].

Connection of the *airframe* object to the *kinematics* object (see Figure 2) makes that the reference systems in both connectors merge, i.e. from then on the airframe is moving freely with respect to the inertial reference, according to kinematic equations described in the *kinematics* object.

The *airframe* object has a second connector on top (see Figure 2). This connector may contain a different reference frame with a constant offset, or may simply be identical to the body frame (to be specified via an offset parameter). It is intended for interconnection of for example external force model components, sensor models, etc. As mentioned before, in case of a flexible airframe also generalised co-ordinates and generalised forces are included in the connector definition.

External forces and moments

The airframe equations of motion are primarily driven

by aerodynamic and propulsion forces and moments. These are computed in corresponding model components in Figure 2. These components often need to be prepared for each aircraft type individually, since application rules and data (sources) behind aerodynamics and propulsion models may strongly differ. For this reason, a base class is available that already defines interfaces and the connector, as well as equations for computation of key variables like the angle of attack, side slip, and true airspeed. Local wind velocities are hereby requested from the *atmosphere* model in Figure 1. The user may develop own model components, inheriting this base class.

Besides the airframe, each component may be developed around its own local reference frame. In case of aerodynamics, these may for example be the stability or wind axes. Interconnection with the airframe follows via a transformation object (e.g. *AeroRef* in Figure 2). This object has two connectors representing two reference systems. The offset (position, orientation) in between may be specified via parameters that become visible and can be edited by double-clicking on the object. The object also relates the forces and moments that act along the connector reference systems. When connecting a model with the *airframe* object, the transformation object makes sure that the kinematics between the local component and the airframe reference systems are correctly related, as well as forces and moments are applied correctly.

The aircraft model in Figure 2 has two aerodynamics models (right hand side). The upper one (*Aero*) contains forces and moments as induced by the over-all motion of the aircraft (“rigid aerodynamics”), usually also corrected for quasi-steady deformation of the airframe. The underlying model may be based on complex application rules, table look-ups, etc. In case a data set is not available or incomplete, computational tools as described in [13] are used. The lower aerodynamics component (*aerodynamicsFlex*) computes unsteady (generalised) forces and moments as induced by flexible deformation of the airframe. For this component extensive pre-processing tools have been developed, involving application of the Doublet-Lattice Method, axis transformations, Rational Function Approximation and removal of quasi-steady effects (already accounted for in the rigid aerodynamics model), see Refs. [19, 13] for more details. The unsteady aerodynamic data are read from a user-specified data file at simulation start.

Note that the *Aero* component is connected to the lower *airframe* connector via the *AeroRef* object,

whereby the latter describes the offset between the airframe body axes and the aerodynamic reference system. The upper aerodynamics component is directly connected with the upper *airframe* connector, making use of generalised co-ordinates declared therein. In case kinematics and the balance between aerodynamic and actuation forces are relevant, a direct interconnection between the *actuators* and aerodynamics models may be added.

The engine models (top left) are connected to the airframe via a slightly different type of transformation. Instead of an offset, the number of a structural grid point, where the object is to be attached, may be specified. At simulation start the transformation object requests the rows of the modal matrix that apply to the grid point from the *airframe* object, allowing it to continuously compute the kinematic relation and force balance between its connectors as a function of the offset from the airframe reference and the local deformation [19, 30]. This for example implies that directional thrust variations due to local deformation at the engine attachment point are automatically taken into account.

Sensor models

The very same principle as used for interconnecting engine models with the airframe structure also applies to the sensor models, located in the top-right corner of Figure 2. A set of sensor types is available in the library. For example, accelerometers compute local accelerations at their point of attachment (specified via grid point number, or offset) as a function of the inertial motion of the airframe, its position in the airframe reference, as well as the local airframe deformation.

The ILS, GPS, and radio altimeter sensors obtain their values by making a function call to the *airport*, *world*, and *terrain* environment models respectively (Figure 1), passing on their momentary inertial position as an argument. In this way, for example multiple GPS sensor objects may be included at various locations on the airframe. Each object can request its very local co-ordinates from the *world* object.

Local wind effects

As already discussed in Section 2.2, mean winds are computed in the atmosphere block at the top level of the model in Figure 1. However, turbulence models are usually described in aircraft body axes, whereby delays as gusts travel along the airframe, are taken into account. This is described in the *localWind* object (lower left, in this case based on EASA CS-AWO specifications for autoland assessment). Random turbulence velocities are obtained from dedicated filters

(Dryden, Karman) that use white noise signals as inputs. This noise is provided via an external connector.

Systems

On-board systems are included in the *actuators* component. This component may describe actuators and hydraulic / electric systems using simple transfer functions, as well as highly detailed physical models, constructed from hydraulics and electronics libraries. The library currently only provides the first variant, since detailed on-board system models are unique for each type or family of aircraft and are usually provided by systems specialists. A recent example of on-board system model implementation using Modelica can be found in [5].

Avionics bus

Finally, the thin bar at the top of Figure 2 represents a so-called *data bus*, implemented using Modelica's expandable connector concept. The data bus includes signals that one would typically find on avionics buses in the aircraft, like the readings of all sensors, command signals to engine and control surface actuators, gear status, etc. For this reason, the sensor, actuator, and engine models have been attached to the bus object. The bus is also accessible from outside and allows direct connection to elements from the Modelica block diagram library. This enables a control system composed using this library to directly communicate with the aircraft data bus.

3 The Flight Dynamics Library

The top-level structure of the DLR Flight Dynamics Library is depicted in Figure 3. The *Modelica* branch in the depicted tree (top) contains the principal standard libraries delivered with Modelica. The branches of the Flight Dynamics Library will be briefly described below:

- **ProjectTemplate** contains a basic library structure for an aircraft. Each aircraft type has its own models for aerodynamics, propulsion, systems, landing gears, etc. These models are built on base classes that already compute all basic variables (e.g. for aerodynamics, angle of attack, calibrated airspeed, etc.) and are stored in this structure. The project template contains a very simple, but readily working aircraft model. The user may copy this template into an own project and start implementing aircraft-specific components,

or add components from the Flight Dynamics or any other Modelica library.

- **Aerodynamics** contains example aerodynamic models for use in rigid and flexible aircraft models, as well as base classes that the user may extend (inherit) to develop his own aircraft-specific model components. Each aircraft type or family namely tends to use unique application rules. For this reason, newly implemented aerodynamics components are stored within the aircraft project (see *ProjectTemplate*).
- **Airframes** contains rigid and flexible airframe model objects. The rigid ones may have constant mass and inertia tensor (entered via parameters), or these may change at a given rate (e.g. as a function of fuel consumption). The flexible airframe component loads its mass, and modal data from an external file (e.g. a Matlab mat-file [21]).
- **Environment** contains all environment-related models as described at the beginning of Section 2.
- **Examples** contains example implementations of various (basic) aircraft models.
- **Gear** currently contains a simplified landing gear model for which basic properties may be set and which may be attached to the *airframe* object in Figure 2. A base class containing a standard interface for interconnection with the *airframe* is provided for implementation of detailed landing gear models, e.g. composed with help of the multi-body library by specialists in the field.
- **Interfaces** contains all library-specific connector types, as well as the data bus that was discussed Section 2.5.
- **Kinematics** contains the *Kinematics* object as described in the previous section.
- **Propulsion** contains, as for the aerodynamics, example engine model implementations, as well as base classes that allow the user to implement his own propulsion models.
- **Systems** mainly contains sensor models (accelerometers, ILS, GPS, etc.) with time constants and noise if desired, and simple transfer function-based actuator models.
- **Transformations** contains standard transformations between reference systems.

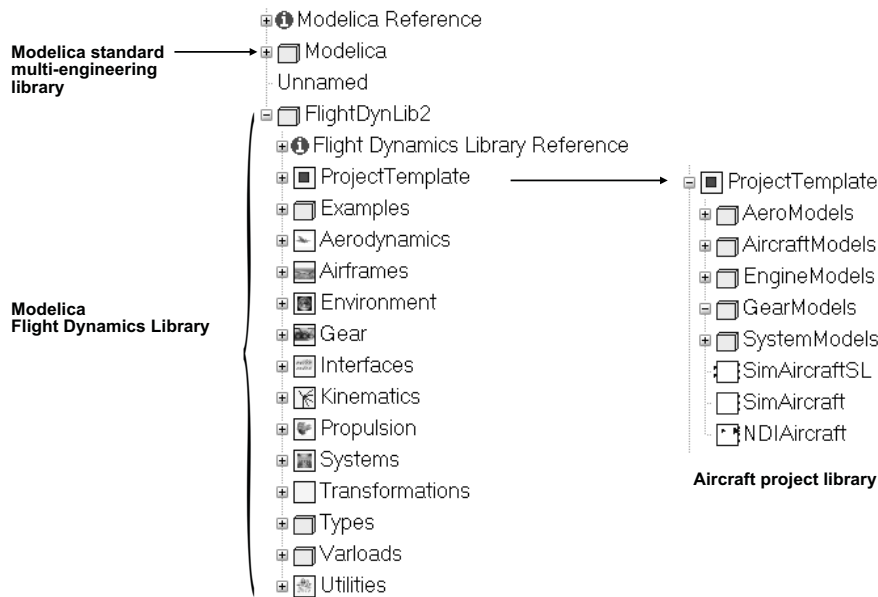


Figure 3: Top-level structure of the Flight Dynamics Library

- **Types** contains type definitions for internal variables, to which the user may add his own.
- **Utilities** contains miscellaneous functions e.g. for reading external data files.

Within a dedicated Modelica modelling and simulation environment, like Dymola (Dynamic Modelling Library [3]) aircraft models may be composed from the library using drag and drop.

4 Automatic code generation

After model composition has been finished, a model translator sorts and solves all model equations according to specified inputs and outputs into Ordinary Differential Equations (ODE's) or Differential Algebraic Equations (DAE's), suitable for use in simulation. A modelling tool that is well capable of doing this is Dymola [3]. Besides a graphical modelling environment and advanced symbolic algorithms, the tool offers extensive simulation and data analysis capabilities. However, the model code may be used in other engineering environments and simulation tools as well, like for example Matlab/Simulink [22]. For this environment an additional tool set has been developed that automatically generates trimming and linearisation scripts, allowing the user to easily specify and accurately compute initial conditions prior to simulation [15].

4.1 Specification of inputs and outputs

A simple way of specifying model inputs and outputs is shown at the top of Figure 1. Here a so-called Avionics block has been connected to the bus connector of the aircraft. At this main model level, also input and output connectors have been defined. The Avionics block injects pilot throttle and control surface input commands (from Throttle, Controls connectors) into the data bus. Output variables of interest, in this case navigation and air data, are read from the bus and passed on to output connectors (*NavOut*, *Airdata*).

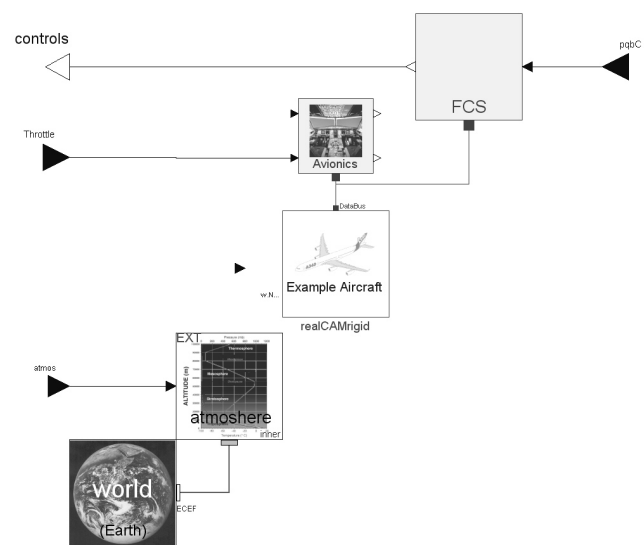


Figure 4: Reversal of inputs and outputs for NDI control law generation

4.2 Inverse model generation

Probably one of the most attractive features of Modelica, in combination with a model compiler like Dymola, is the possibility to generate inverse models just as easily as normal simulation models. Inverse models are extremely useful for fast and accurate trim computation, systems and control surface sizing, as well as for automatic generation of control laws that are based on inverse model equations, like Nonlinear Dynamic Inversion (NDI [9]). Ref. [18] describes various types of inverse model-based control laws and their automatic generation from Modelica models. Figure 4 shows the addition of an FCS block, which contains the basic structure for an NDI controller, see Figure 5. Like the *Avionics* block, the FCS communicates with the aircraft model via the expandable bus. Compared with Figure 1, the *Controls* input has become an output, and command variables $pqbC$ (roll rate p_b , pitch rate q_b , side slip angle β) have been added as inputs instead. This is basically all that is needed to generate an inverse model. For practical reasons, additional modifications have been made, like the removal of most environment models: variables like radio altitude and ambient pressure can be directly obtained from measurement instead.

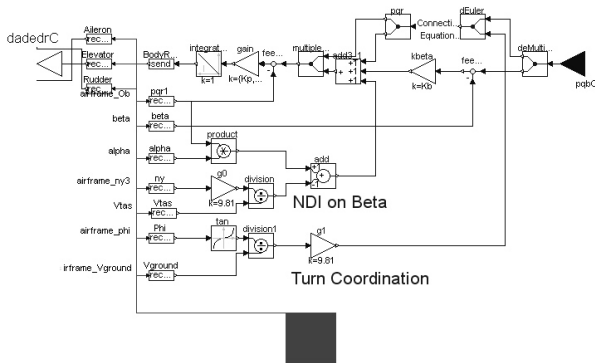


Figure 5: NDI control law with command variables for manual aircraft control

4.3 Desktop visualisation

Easy access to high-quality desktop visualisation tools becomes more and more important in the flight control law design process. This helps the engineer to better understand the (closed-loop) dynamics of the aircraft, and allows her or him to interactively "fly" the aircraft to qualitatively assess control law performance and to find weaknesses before implementation in the full flight simulator. One of the first commercial environments offering this visualisation capability is the

Aviator Visual Design Simulator (AVDS), described in Ref. [29]. An interface with the Flight Dynamics Library has been described in [24]. Interfaces with public domain simulator programs like FlightGear [2] are to be developed, whereby the internal flight dynamics model in this program is overwritten by the model constructed from the Flight Dynamics Library.

5 Application examples

Since its first version in 1995, the Flight Dynamics Library has been applied in several projects at DLR, especially involving model development for design and evaluation of flight control laws. A number of these applications will be briefly discussed in this section.

REAL – Automatic Landing

In the frame of the EU-project REAL (Robust and Efficient Auto pilot control Laws design [31]) for the first time inverse model equations for a transport aircraft were automatically generated from the model implementation in Modelica. These inverse equations were used as the core of an automatic landing system that was developed in the frame of this project [16]. The control laws were successfully flight tested on DLR's test bed ATTAS (Advanced Technologies Testing Aircraft System [6]) during six automatic landings.

X-31A with reduced vertical tail

The same procedure for automatic generation of Nonlinear Dynamic Inversion control laws was applied to the thrust-vectorred experimental fighter aircraft X-31A in the frame of the project VECTOR (Vectoring, Extremely short take-off and landing, Control, Tail-less Operations Research [12]), in order to investigate reduced vertical tail configurations of this aircraft [32]. The control laws were exported from Dymola and implemented in the ground-based flight simulator in Patuxent River, MD, USA, and successfully evaluated by five test pilots and one fleet pilot.

3D-Flexible aircraft flight simulator

From the example flexible aircraft model as presented in Section 2 simulation code was generated for use in interactive real-time simulation. The model was augmented with automatically generated Dynamic Inversion-based control laws, the automatic landing system as developed in the project REAL, as well as load alleviation control laws. An engineering visualisation environment called *VisEngine*, developed by AeroLabs AG [1], simultaneously visualises aircraft flight dynamics and structural dynamics in real-time

in very high quality, see Figure 6. As exclusive features for DLR, VisEngine was extended with on-line visualisation of airframe deformation, as well as the capability of visualising the aircraft and its environment using 3-D stereo projection. The visualisation of structural deformation greatly helped to qualitatively assess performance of structural control laws.

GARTEUR FM(AG17) Control law design for aircraft on ground

In the future, transport aircraft may be equipped with drive-by-wire control laws to reduce pilot workload during landing roll-out, taxiing, and take-off. In addition, this technology will provide the basis for development of functions for automatic ground manoeuvring [8]. In the frame of this GARTEUR project a complex aircraft model, including landing gears, was implemented using the Flight Dynamics Library. In order to study various possible command variables for the pilot, automatic model inversion was used to develop control laws based on nonlinear dynamic inversion. More details can be found in Refs. [19, 17].



Figure 6: 3D-stereo visualisation of aircraft flight and structural dynamics, and aerodynamic loads

6 Conclusions

In this paper an overview of the Modelica-based Flight Dynamics Library has been given. This library allows for intuitive construction of multi-disciplinary models for use in the aircraft and flight control laws design process. To this end, the library offers the following unique features:

- full compatibility with other libraries based on the Modelica language, allowing for development of truly multi-disciplinary aircraft models on a common modelling platform;
- intuitively structured models due to a physics-oriented break-down into model components and

interactions;

- construction of rigid just as well as fully flexible aircraft models, including unsteady aerodynamic effects;
- easy implementation of multiple aircraft models using the same set of environment models (earth, atmosphere, etc.);
- automatic generation of efficient simulation code for various engineering environments (using a Modelica tool like Dymola);
- automatic generation of inverse model code, e.g. for use in nonlinear control laws;
- automatic generation of trimming and linearisation scripts for use with the model in Matlab/Simulink;
- easy integration with desktop visualisation.

References

- [1] Aerolabs AG – Professional Solutions for the Engineering Industry: <http://www.aerolabs.de>.
- [2] Flight Gear web site: <http://www.flightgear.org>.
- [3] Dynasim AB. *Dymola – User’s Manual*. <http://www.dynasim.se/>, 1994–2006.
- [4] anon. Department of Defence World Geodetic System 1984 – Its Definition and Relationships with Local Geodetic Systems, Third Edition, Amendment 1. Technical Report NIMA TR8350.2, National Imagery and Mapping Agency, Bethesda, MD, January 2000. Third Edition.
- [5] J. Bals, G. Hofer, A. Pfeiffer, and C. Schallert. Virtual Iron Bird – A Multidisciplinary Modelling And Simulation Platform For New Aircraft System Architectures. In *DGLR Luft- und Raumfahrtkongress 2005, Friedrichshafen, DGLR-Jahrbuch 2005*, 2005.
- [6] M. Bauschat, W. Mönnich, D. Willemsen, and G. Looye. Flight Testing Robust Autoland Control Laws. In *Proceedings of the AIAA Guidance, Navigation and Control Conference 2001, Montreal CA*, 2001.
- [7] Rudolf Brockhaus. *Flugregelung*. Springer-Verlag, Berlin Heidelberg, 1994.
- [8] J. Duprez, F. Mora-Camino, and F. Villaumé. Control of the Aircraft-on-Ground Lateral Motion During Low Speed Roll and Manoeuvres. In *IEEE Aerospace Conference Proceedings*, pages 2656–2666, 2004.
- [9] Dale Enns, Dan Bugajski, Russ Hendrick, and Gunter Stein. Dynamic Inversion: An Evolving Methodology for Flight Control Design. In *AGARD Conference Proceedings 560: Active Control Technology: Applications and Lessons Learned*, pages 7–1 – 7–12, Turin, Italy, May 1994. NATO-AGARD.

- [10] European Aviation Safety Agency (EASA). Certification Specifications for All Weather Operations CS-AWO, 2003. Available from http://www.easa.europa.eu/home/certspecs_en.html.
- [11] Christopher Fielding, Andras Varga, Samir Bennani, and Michiel Selier (Eds.). *Advanced Techniques for Clearance of Flight Control Laws*. Lecture Notes in Control and Information Sciences 283. Springer Verlag, London, 2002.
- [12] H. Friehmelt and P. Huber. Vector- Die X31A fliegt zu neuen bahnbrechenden Technologiedemonstrationen. In *DGLR Luft- und Raumfahrtkongress 2001, Hamburg 17.-20. Sept. 2001*.
- [13] Thiemo Kier, Gertjan Looye, Moriz Scharpenberg, and Marion Reijerkerk. Process, methods and tools for flexible aircraft flight dynamics model integration. In *Proceedings of the International Forum on Aeroelasticity and Structural Dynamics (IFASD)*, 2007.
- [14] F. G. Lemoine, S. C. Kenyon, J. K. Factor, R.G. Trimmer, N. K. Pavlis, D. S. Chinn, C. M. Cox, S. M. Klosko, S. B. Luthcke, M. H. Torrence, Y. M. Wang, R. G. Williamson, E. C. Pavlis, R. H. Rapp, and T. R. Olson. The Development of the Joint NASA GSFC and NIMA Geopotential Model EGM96. Technical Report NASA/TP-1998-206861, NASA Goddard Space Flight Center, Greenbelt, Maryland, July 1998.
- [15] Gertjan Looye. Integrated Flight Mechanics and Aeroelastic Aircraft Modeling using Object-Oriented Modeling Techniques. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference*, Portland, USA, August 1999. AIAA-99-4192.
- [16] Gertjan Looye. Design of Robust Autopilot Control Laws with Nonlinear Dynamic Inversion. *at – Automatisierungstechnik*, 49(12), 2001.
- [17] Gertjan Looye. *Rapid prototyping using inversion-based control and object-oriented modelling.*, chapter 8. Lecture Notes in Control and Information Sciences. Springer Verlag, Berlin, 2007.
- [18] Gertjan Looye, Michael Thümmel, Matthias Kurze, Martin Otter, and Johann Bals. Nonlinear Inverse Models for Control. In *Proceedings of the third international Modelica conference*, Hamburg, March 2005.
- [19] Gertjan H.N. Looye. *An Integrated Approach to Aircraft Modelling and Flight Control Law Design*. 2008. Doctoral thesis Delft University of Technology. Available from <http://repository.tudelft.nl>.
- [20] Jean-François Magni, Samir Bennani, and Jan Terlouw (Eds.). *Robust Flight Control – A Design Challenge*. Lecture Notes in Control and Information Sciences 224. Springer Verlag, London, 1997.
- [21] The Math Works Inc. *MATLAB – External Interfaces Reference – Version 7*, March 2005.
- [22] The Math Works Inc. *Simulink Reference*, 2005.
- [23] Susan McLean, Susan Macmillan, Stefan Maus, Vincent Lesur, Alan Thomson, and David Dater. The US/UK World Magnetic Model for 2005-2010. Technical Report NOAA Technical Report NESDIS/NGDC-1, NOAA National Geophysical Data Center, British Geological Survey Geomagnetism Group, December 2004.
- [24] D. Moormann and G. Looye. The modelica flight dynamics library. In *Proceedings of the 2nd International Modelica Conference*, pages 275–284. The Modelica Association, 2002. Available from <http://www.modelica.org>.
- [25] D. Moormann, P.J. Mosterman, and G. Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, 3(3), April 1999.
- [26] Dieter Moormann. Physical modeling of controlled aircraft. In *Proceedings of the CESA'96 IMACS Multiconference on Computational Engineering in Systems Applications*, pages 970–975, Lille-France, July 1996.
- [27] Dieter Moormann. *Automatisierte Modellbildung der Flugsystemdynamik*. VDI Verlag, Reihe 8, Dissertation RWTH Aachen, Institut für Flugdynamik, Düsseldorf, 2002.
- [28] Martin Otter, Hilding Elmquist, and Sven Erik Mattsson. The new modelica multibody library. In Peter Fritzson, editor, *Proceedings of the 3rd International Modelica Conference, Linköping, Sweden*, pages 311 – 330. Modelica Association, 2003.
- [29] S.J. Rasmussen and S.G. Breslin. AVDS: a Flight Systems Design Tool for Visualization and Engineer-in-the-Loop Simulation. AIAA 97-3467. 1997.
- [30] Christian Reschke. Flight loads analysis with inertially coupled equations of motion. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, 2005. AIAA 2005-6026.
- [31] W.F.J.A. Rouwhorst. Robust and Efficient Autopilots control Laws design, demonstrating the use of modern robust control design methodologies in the autoland system design process – the REAL Project. In *Proceedings of the Aeronautics Days 2001, Hamburg Germany*, January 2001.
- [32] R. Steinhäuser, G. Looye, and O. Brieger. Design and Evaluation of Control Laws for the X-31A with Reduced Vertical Tail. In *Proceedings of the AIAA Guidance and Control Conference*, Providence, Rhode Island, USA, August 2004. AIAA-2004-5031.
- [33] Brian L. Stevens and Frank L. Lewis. *Aircraft Control and Simulation*. Wiley-Interscience Publication. John Wiley & Sons, Inc., New York, 2000.
- [34] Martin R. Waszak and Dave K. Schmidt. Flight Dynamics of Aeroelastic Vehicles. *AIAA Journal of Aircraft*, 25(6):563–571, June 1988.

Implementation of the Hertz Contact Model and Its Volumetric Modification on Modelica

Ivan Kosenko Evgeniy Alexandrov

Russian State University of Tourism and Service, Department of Engineering Mechanics
Glavnaya str. 99, Cherkizovo-1, Moscow reg., 141221, Russia

Abstract

The Hertz model of an elastic bodies contact and its volumetric modification are analyzed for the proper implementation on Modelica. Computational algorithms applied aim to accelerate the simulation process and make it more reliable.

The algorithm tracking the surfaces of the bodies which are able to contact was improved using its differential version and showed an accuracy high enough. Simulation of the Hertz model was accelerated due to use of the differential technique to compute the complete elliptic integrals and due to the replacement of the implicit transcendental equation by the differential one.

To have a reliable model for the simulation of the contact especially in case of the contact spot ellipses of an eccentricity high enough the volumetric modification of Hertz model is introduced. The model showed a reliable behavior and an acceptable accuracy.

Finally an implementation of the ball bearing model as an example of the contact models application is under consideration. The particular bearings being analyzed can have different number of balls and different types of raceways. The bearing models created using the library of classes developed earlier and have an outside look exactly like a mechanical constraints and behave in some degree similar to the revolute joints.

Keywords: Hertz contact model, volumetric approach, ball bearing model

1 Introduction

It is known [1] to compute a force of the elastic bodies interaction at a contact several different approaches are applied: (a) the classical Hertz model [2], (b) the model based on the polygonal approximation of the contacting surfaces [3] applied to cases of the surfaces of a complex shape and implemented on Modelica [4], (c) the volumetric model [1, 5]. In our model we fol-

low the classical Hertz approach, and the normal force computation method is a main topic of our analysis. To handle with the surfaces at the contact we apply an approach mentioned in [4] as variant 2: algebraic constraint surfaces, which we frequently use in our models. For definiteness and simplicity to simulate the tangent contact force one uses a regularized model of the Coulomb friction [6]. This is sufficient enough to simulate the dynamics over time of the machine under simulation lifecycle. May be some additional complications for the friction model, e. g. an account of the lubrication of any type, will be needed.

2 Reduction in Vicinity of Contact

Keeping a frame of the formalism applied previously to simulate a unilateral constraint [6] consider its particular case corresponding to mechanics of contact interaction for two elastic bodies, identified hereafter as A and B . Their outer surfaces, see Figure 1, being at contact supposed sufficiently regular.

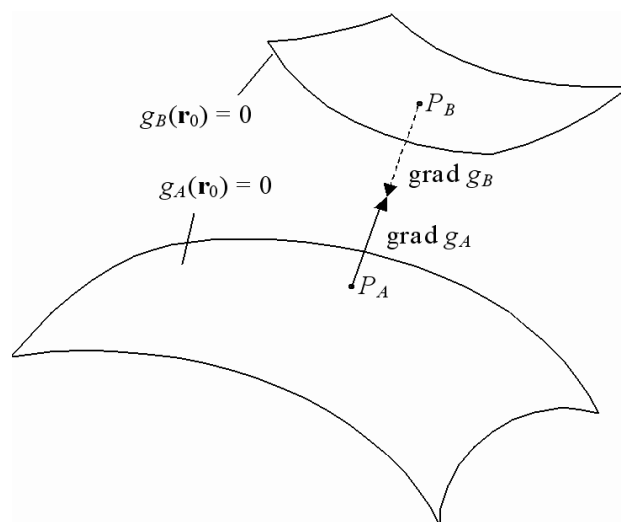


Figure 1: Vicinity of the Contact Area

Applying the same notations as ones used in [6] we

start here by reproducing the system of eight scalar algebraic equations

$$\begin{aligned} \text{grad } g_A(\mathbf{r}_{P_A}) &= \lambda \cdot \text{grad } g_B(\mathbf{r}_{P_B}), \\ \mathbf{r}_{P_A} - \mathbf{r}_{P_B} &= \mu \cdot \text{grad } g_B(\mathbf{r}_{P_B}), \\ g_A(\mathbf{r}_{P_A}) &= 0, \\ g_B(\mathbf{r}_{P_B}) &= 0. \end{aligned} \quad (1)$$

defining the coordinates $x_{P_A}, y_{P_A}, z_{P_A}, x_{P_B}, y_{P_B}, z_{P_B}$ of the outer surfaces opposing points P_A, P_B , see Figure 1. Here the coordinate vectors $\mathbf{r}_{P_A} = (x_{P_A}, y_{P_A}, z_{P_A})^T$, $\mathbf{r}_{P_B} = (x_{P_B}, y_{P_B}, z_{P_B})^T$ are defined with respect to (w. r. t.) the absolute coordinate frame $O_0x_0y_0z_0$ of reference (AF) usually connected to the multibody system base body B_0 . Note the functions $g_A(\mathbf{r}_0) = g_A(\mathbf{r}_0, t)$, $g_B(\mathbf{r}_0) = g_B(\mathbf{r}_0, t)$ are really a time dependent ones, and define the outer surfaces current spatial position of the bodies at a contact w. r. t. AF . The values λ, μ are an auxiliary variables.

It turned out by the computational practice with Dymola the most suitable approach to implement a system of algebraic equations like (1) is to replace it by the system of DAEs properly derived from (1). It can be done by introducing an additional variables which the time derivatives and thus compose the differential subsystem

$$\dot{\mathbf{r}}_{P_A} = \mathbf{u}_{P_A}, \quad \dot{\mathbf{r}}_{P_B} = \mathbf{u}_{P_B}, \quad \dot{\lambda} = \xi, \quad \dot{\mu} = \eta, \quad (2)$$

completed by the algebraic one

$$\begin{aligned} [\boldsymbol{\omega}_A, \text{grad } g_A] + T_A \text{Hess } f_A T_A^T (\mathbf{u}_{P_A} - \mathbf{v}_{P_A}) - \xi \text{grad } g_B - \\ \lambda ([\boldsymbol{\omega}_B, \text{grad } g_B] + T_B \text{Hess } f_B T_B^* (\mathbf{u}_{P_B} - \mathbf{v}_{P_B})) &= \mathbf{0}, \\ \mathbf{u}_{P_A} - \mathbf{u}_{P_B} - \eta \text{grad } g_B - \\ \mu ([\boldsymbol{\omega}_B, \text{grad } g_B] + T_B \text{Hess } f_B T_B^T (\mathbf{u}_{P_B} - \mathbf{v}_{P_B})) &= \mathbf{0}, \\ (\text{grad } g_A, \mathbf{u}_{P_A}) - (\text{grad } f_A, T_A^T \mathbf{v}_{P_A}) &= 0, \\ (\text{grad } g_B, \mathbf{u}_{P_B}) - (\text{grad } f_B, T_B^T \mathbf{v}_{P_B}) &= 0. \end{aligned} \quad (3)$$

where the vectors $\mathbf{v}_{P_A}, \mathbf{v}_{P_B}$ are a velocities of the bodies physical points currently located at the geometric points P_A, P_A and are to be calculated according to the Euler formula

$$\mathbf{v}_{P_\alpha} = \mathbf{v}_{O_\alpha} + [\boldsymbol{\omega}_\alpha, \mathbf{r}_{P_\alpha} - \mathbf{r}_{O_\alpha}] \quad (\alpha = A, B),$$

where O_A, O_B are the bodies masscenters, $\boldsymbol{\omega}_A, \boldsymbol{\omega}_B$ are the angular velocities of the bodies. Matrices $\text{Hess } f_A, \text{Hess } f_B$ are the Hesse ones of the functions f_A, f_B defining the bodies outer surfaces w. r. t. the bodies central principal coordinate systems. The the functions f_A, f_B relate to the ones g_A, g_B according to the equations

$$g_\alpha(\mathbf{r}_0) = f_\alpha [T_\alpha^T (\mathbf{r}_0 - \mathbf{r}_{O_\alpha})] \quad (\alpha = A, B),$$

and their gradients connected by

$$\text{grad } g_\alpha(\mathbf{r}_{P_\alpha}) = T_\alpha \text{grad } f_\alpha [T_\alpha^T (\mathbf{r}_{P_\alpha} - \mathbf{r}_{O_\alpha})],$$

where T_A, T_B are the orthogonal matrices defining current orientation of the bodies.

Surely, in case of the DAEs use one has to provide a consistent initial values for all the additional state variables introduced here for each object of the compliant contact being under construction in the sequel.

Unlike our previous approach [6] now we suppose the bodies A and B don't create any obstacles for their relative motion. If 3D-regions bounded by the bodies outer surfaces don't intersect then the object of a constraint, rather of a contact, generates a zero wrench in the direction of each body. Simultaneously this object has to generate the radius vectors $\mathbf{r}_{P_A}, \mathbf{r}_{P_B}$ of opposing with each other points P_A, P_B .

Based on (1) note the variable μ indicates the contact of the bodies A and B . Indeed, for definiteness suppose the outer surfaces in vicinities of the points P_A, P_B are such that vectors of gradients $\text{grad } g_A(\mathbf{r}), \text{grad } g_B(\mathbf{r})$ are directed outside the each body. Then we have the following cases at hand: (a) $\mu > 0$ means the contact absent; (b) $\mu \leq 0$: the contact takes place. If $\mu < 0$ then the bodies supposed to penetrate each other, though really begin to deform in a region of the contact. In the sequel we follow the simplest elastic contact model originating from Hertz [2]. Computational analysis will be performed for the case of contacting only, see Figure 2. For simplicity and definiteness the surfaces are showed convex in Figure 2 though it is not necessary at all in general for our implementation.

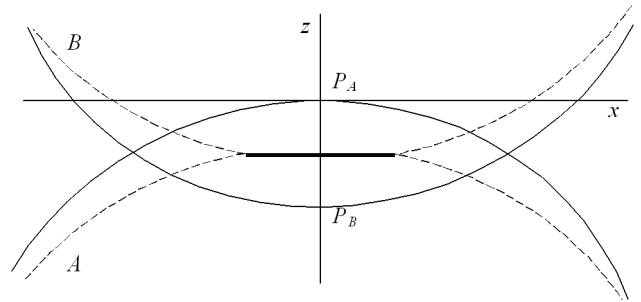


Figure 2: Local Coordinate System

To represent the Hertz contact model in its classical form first of all we have to construct an auxiliary base in vicinity of the contact. First base is composed by three unit vectors $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ such that $\boldsymbol{\gamma} = \mathbf{n}_A$, where \mathbf{n}_A is the unit vector along the gradient $\text{grad } g_A(\mathbf{r})$ collinear to the z -axis in Figure 2. As it was for the derivation of the opposing points the most appropriate move to compute the proper base $\{\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}\}$ is to construct a relevant

subsystem of DAEs. First of all start with differential equation for $\boldsymbol{\gamma}$. It has the form

$$\dot{\boldsymbol{\gamma}} = |\text{grad } g_A|^{-1} [(\text{grad } g_A)^\cdot - (\mathbf{n}_A, (\text{grad } g_A)^\cdot) \mathbf{n}_A],$$

where the time derivative of the gradient reads

$$(\text{grad } g_A)^\cdot = [\boldsymbol{\omega}_A, \text{grad } g_A] + T_A \text{Hess } f_A T_A^T (\mathbf{u}_{P_A} - \mathbf{v}_{P_A}).$$

Now we can right down the chain of equations

$$\boldsymbol{\Omega} = [\boldsymbol{\gamma}, \dot{\boldsymbol{\gamma}}], \quad \dot{\boldsymbol{\alpha}} = [\boldsymbol{\Omega}, \boldsymbol{\alpha}], \quad \dot{\boldsymbol{\beta}} = [\boldsymbol{\gamma}, \boldsymbol{\alpha}],$$

defining successively the angular velocity $\boldsymbol{\Omega}$ of the unit vector $\boldsymbol{\gamma}(t)$ rotation, the differential equation for the unit vector $\boldsymbol{\alpha}$, and the unit vector $\boldsymbol{\beta}$ completing the local base under construction. Actually the vector $\boldsymbol{\Omega}$ is an angular velocity of the base triple $\{\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}\}$ w. r. t. AF .

Note once more we have to provide the consistent initial data for the vectors $\boldsymbol{\alpha}, \boldsymbol{\gamma}$ which became now a vector valued state variables. And of course browsing the equations represented above it is transparent enough which Modelica code stands behind the algorithm outlined here.

Using the base $\{\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}\}$ built up above it is easy enough to compose the matrix $T = [\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}]$ consisting of the columns composed themselves by the coordinates of the unit vectors. Actually T is the transfer matrix between coordinates of AF and the current local base $\{\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}\}$. Let us first express the outer surfaces equations in coordinates of the local system (LF) having an origin at the point P_A , see Figure 2.

Because the matrix T is orthogonal its inverse is derived by the transposition of T . Then to compute the matrix of coordinate transformation from the LF to one of the bodies' we can represent it as follows

$$R_\alpha = T^T T_\alpha \quad (\alpha = A, B).$$

Introducing new temporary notation $\mathbf{r} = (x, y, z)^T$ for the coordinate vector of the current geometric point w. r. t. local system P_Axyz one can easily deduce the dependence

$$\mathbf{r} = \boldsymbol{\rho}_{O_\alpha} + R_\alpha \mathbf{r}_\alpha \quad (\alpha = A, B),$$

where $\boldsymbol{\rho}_{O_\alpha} = (\xi_{O_\alpha}, \eta_{O_\alpha}, \zeta_{O_\alpha})^T$ is the coordinate vector of the body α mass center w. r. t. LF .

Let the body α outer surface is defined by the equation

$$h_\alpha(\mathbf{r}) = 0 \quad (\alpha = A, B), \quad (4)$$

w. r. t. current position of LF . Then it is easy to see the function h_α can be computed by the formula

$$h_\alpha(\mathbf{r}) = f_\alpha(R_\alpha^T (\mathbf{r} - \boldsymbol{\rho}_{O_\alpha})) \quad (\alpha = A, B).$$

Further bring the equations (4) to the form suitable to analyse the contact problem in vicinity of the corresponding points

$$\boldsymbol{\rho}_{P_\alpha} = \boldsymbol{\rho}_{O_\alpha} + R_\alpha \boldsymbol{\rho}_\alpha \quad (\alpha = A, B),$$

on the surface of the body α . Here the vector $\boldsymbol{\rho}_\alpha$ defines the position of the point P_A w. r. t. the body α own coordinate system. Supposing the surfaces regular enough we have the expansions

$$\begin{aligned} f_\alpha(\mathbf{r}_\alpha) &= f_\alpha(\boldsymbol{\rho}_\alpha) + (\text{grad } f_\alpha(\boldsymbol{\rho}_\alpha), \Delta \mathbf{r}_\alpha) + \\ &\quad \frac{1}{2} (\text{Hess } f_\alpha(\boldsymbol{\rho}_\alpha) \Delta \mathbf{r}_\alpha, \Delta \mathbf{r}_\alpha) + O(|\Delta \mathbf{r}_\alpha|^3), \\ h_\alpha(\mathbf{r}) &= h_\alpha(\boldsymbol{\rho}_{P_\alpha}) + (\text{grad } h_\alpha(\boldsymbol{\rho}_{P_\alpha}), \Delta \mathbf{r}) + \\ &\quad \frac{1}{2} (\text{Hess } h_\alpha(\boldsymbol{\rho}_{P_\alpha}) \Delta \mathbf{r}, \Delta \mathbf{r}) + O(|\Delta \mathbf{r}|^3), \end{aligned}$$

where $\Delta \mathbf{r} = \mathbf{r} - \boldsymbol{\rho}_{P_\alpha}$, $\Delta \mathbf{r}_\alpha = \mathbf{r}_\alpha - \boldsymbol{\rho}_\alpha$. Since $\Delta \mathbf{r} = R_\alpha \Delta \mathbf{r}_\alpha$ then it is easy to verify that

$$\begin{aligned} \text{grad } h_\alpha(\boldsymbol{\rho}_{P_\alpha}) &= R_\alpha \text{grad } f_\alpha(\boldsymbol{\rho}_\alpha) \\ \text{Hess } h_\alpha(\boldsymbol{\rho}_{P_\alpha}) &= R_\alpha \text{Hess } f_\alpha(\boldsymbol{\rho}_\alpha) R_\alpha^T. \end{aligned} \quad (5)$$

Because at the body α outer surface point $\boldsymbol{\rho}_{P_\alpha}$ the function h_α is zero-valued then up to the terms of the third order in the coordinate system $P_\alpha xyz$ the equation (4) can be represented as follows

$$\frac{\partial h_\alpha}{\partial z} z + \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a_\alpha & c_\alpha \\ c_\alpha & b_\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = 0, \quad (6)$$

where the Hesse matrix elements are to be expressed by the formulae

$$a_\alpha = \frac{1}{2} \frac{\partial^2 h_\alpha}{\partial x^2}, \quad b_\alpha = \frac{1}{2} \frac{\partial^2 h_\alpha}{\partial y^2}, \quad c_\alpha = \frac{1}{2} \frac{\partial^2 h_\alpha}{\partial x \partial y},$$

where in turn one should use the results of (5). Note the equation (6) has such a simple representation because at the point $\boldsymbol{\rho}_{P_\alpha}$ the choice of the base causes the conditions

$$\frac{\partial h_\alpha}{\partial x}(\boldsymbol{\rho}_{P_\alpha}) = 0, \quad \frac{\partial h_\alpha}{\partial y}(\boldsymbol{\rho}_{P_\alpha}) = 0. \quad (7)$$

Supposing the surfaces are nondegenerate at the points P_α we have the condition

$$|\text{grad } h_\alpha(\mathbf{r}_\alpha)| > 0,$$

and because of (7) it causes the condition

$$\frac{\partial h_\alpha}{\partial z}(\mathbf{r}_\alpha) \neq 0.$$

Therefore, the equation (6) can be resolved w. r. t. the variable z in explicit form as

$$z = a'_\alpha x^2 + 2c'_\alpha xy + b'_\alpha y^2, \quad (8)$$

where the new coefficients of the second order terms are computed in the form

$$a'_\alpha = -\frac{a_\alpha}{\frac{\partial h_\alpha}{\partial z}}, \quad b'_\alpha = -\frac{b_\alpha}{\frac{\partial h_\alpha}{\partial z}}, \quad c'_\alpha = -\frac{c_\alpha}{\frac{\partial h_\alpha}{\partial z}}.$$

The further reduction comes to a transformation to canonical view of the quadratic form

$$q(x, y) = ax^2 + 2cxy + by^2, \quad (9)$$

derived as a difference between the forms (8) such that

$$a = a'_B - a'_A, \quad b = b'_B - b'_A, \quad c = c'_B - c'_A.$$

The transformation is implemented simply as a rotation about the z -axis of the system P_Axy to achieve the coefficient c vanishes. Finally the function (9) becomes having the form

$$q(x, y) = Px^2 + Qy^2 \quad (10)$$

with the additional condition $0 < P \leq Q$.

3 The Hertz Model

According to the known technique [7] to compute the total normal force at the contact we have to solve the system

$$\begin{aligned} \frac{FD}{\pi} \int_0^\infty \frac{d\xi}{\sqrt{(\alpha + \xi)(\beta + \xi)\xi}} &= h, \\ \frac{FD}{\pi} \int_0^\infty \frac{d\xi}{(\alpha + \xi)\sqrt{(\alpha + \xi)(\beta + \xi)\xi}} &= P, \\ \frac{FD}{\pi} \int_0^\infty \frac{d\xi}{(\beta + \xi)\sqrt{(\alpha + \xi)(\beta + \xi)\xi}} &= Q, \end{aligned} \quad (11)$$

of three transcendental equations provided the coefficients P , Q from the representation (10) and depth of mutual penetration, so-called mutual approach, $h = |\mathbf{r}_{P_B} - \mathbf{r}_{P_A}|$ are already have been computed. The system (11) has three unknown variables: α , β , F , where the values α , β are the semi-major axes squared of the contact spot ellipse, and F is the total normal elastic force really distributed over the contact area. The parameter

$$D = \frac{3}{4} \left(\frac{1 - \nu_A^2}{E_A} + \frac{1 - \nu_B^2}{E_B} \right)$$

summarizes elastic properties of the contacting bodies: ν_A , ν_B being Poisson's ratios, and E_A , E_B being corresponding Young's moduli.

Using the substitution $\xi \mapsto \eta$ ($\xi = \lambda\eta$) in elliptic integrals of (11) we can separate the last two equations of (11). Indeed, introducing new scaled unknown variables α' , β' according to formulae $\alpha' = \alpha/\lambda$, $\beta' = \beta/\lambda$ we can deduce the two mentioned equations to the closed system

$$I_1(\alpha', \beta') = P, \quad I_1(\beta', \alpha') = Q, \quad (12)$$

if the scaling factor λ satisfies the norming condition

$$\frac{FD}{\pi} \cdot \frac{1}{\lambda^{3/2}} = 1. \quad (13)$$

Here the elliptic integral $I_1(\alpha, \beta)$ is defined by

$$I_1(\alpha, \beta) = \int_0^\infty \frac{d\xi}{(\alpha + \xi)\sqrt{(\alpha + \xi)(\beta + \xi)\xi}}$$

causing clearly verified equations

$$I_1(\alpha', \beta') = -2 \frac{\partial I(\alpha', \beta')}{\partial \alpha'}, \quad I_1(\beta', \alpha') = -2 \frac{\partial I(\alpha', \beta')}{\partial \beta'}, \quad (14)$$

where taking into account that $\alpha' \geq \beta'$, which is equivalent to the condition $P \leq Q$ satisfied above, we may have the relations

$$I(\alpha', \beta') = \int_0^\infty \frac{d\xi}{\sqrt{(\alpha' + \xi)(\beta' + \xi)\xi}} = \frac{2}{\sqrt{\alpha'}} K(k),$$

where in turn $K(k)$ is the complete elliptic integral of the first kind with the modulus defined by the formula

$$k = \sqrt{\frac{\alpha' - \beta'}{\alpha'}}.$$

Here one can see the value k actually has a geometric sense exactly of the contact spot ellipse eccentricity. Using the work [8] as a pattern we introduce the value $c = k^2$ of the elliptic integral modulus square. Taking into account that elliptic integrals are regular functions of $c = 1 - \beta'/\alpha'$ we obtain using the rule of the compound function differentiation

$$\begin{aligned} I_1(\alpha', \beta') &= \frac{2}{\alpha'^{3/2}} \left(K(c) - 2(1 - c) \frac{dK(c)}{dc} \right), \\ I_1(\beta', \alpha') &= \frac{4}{\alpha'^{3/2}} \frac{dK(c)}{dc}. \end{aligned}$$

Dividing then the first equation of (12) by the second one and using the last derived expressions we reduce finally the whole problem to the one-dimensional transcendental equation

$$\frac{1}{2} K(c) \left(\frac{dK(c)}{dc} \right)^{-1} - (1 - c) = \frac{P}{Q} \quad (15)$$

w. r. t. the unknown value c .

Once the solution of the equation (15) had been found we can obtain immediately the values

$$\alpha' = \left(\frac{4}{Q} \frac{dK(c)}{dc} \right)^{2/3} \quad \beta' = \alpha'(1-c).$$

Using the first equation of (11) and normalizing dependence (13) we then find the value of the scaling factor

$$\lambda = \frac{h}{I(\alpha', \beta')} \quad (16)$$

thus arriving to the Hertz problem solution: the normal force and the contact ellipse semi-major axes values

$$F = \frac{\pi}{D} \lambda \sqrt{\lambda}, \quad a = \sqrt{\lambda \alpha'}, \quad b = \sqrt{\lambda \beta'}.$$

Nevertheless an implementation on Dymola requires a further reduction of the model in a manner we already mentioned above twice: use preferably the differential equations (probably to overcome the potential problems for the analytical processor when differentiating the transcendental expressions on the DAE system index reduction stage when compiling and indirectly and more rarely when running the model). To this end we have to remind the known ODEs connecting the complete elliptic integrals of the first $K(c)$ and the second $E(c)$ kind between one another [8]

$$\frac{dK}{dc} = \frac{E - (1-c)K}{2c(1-c)}, \quad \frac{dE}{dc} = \frac{E - K}{2c}.$$

Furthermore instead of (15) then we should use its differential version

$$\left[3 \left(\frac{dK}{dc} \right)^2 - K \frac{d^2K}{dc^2} \right] \dot{c} = 2 \left(\frac{dK}{dc} \right)^2 \dot{C},$$

where $C = P/Q$, and

$$\frac{d^2K}{dc^2} = \frac{(1-c)(2-3c)K - (2-4c)E}{4c^2(1-c)^2}.$$

In this way the complete integrals become an additional state variables such that

$$\dot{K} = \frac{dK}{dc} \dot{c}, \quad \dot{E} = \frac{dE}{dc} \dot{c},$$

and simultaneously we have yet another way to compute elliptic integrals in dynamics, note: exclusively fast and sufficiently accurate way.

Staying in frame of the traditional Hertz model and taking into account that the expression for the normal force has the form

$$F_{\text{elast}} = -e(P, Q)h^{3/2},$$

where while changing the value h the values P, Q don't change, we conclude the potential energy of elastic deformations is represented by the expression

$$U_{\text{elast}} = \frac{2}{5} e(P, Q) h^{5/2}.$$

On the other hand using the volumetric approach [5] one can try to represent the same potential energy as follows

$$U_{\text{elast}} = f \left(\frac{b}{a} \right) V^\nu S^\sigma p^\delta,$$

where V is the volume of the bodies undeformed material intersected, S is the area of the intersection projection onto the xy -plane of the LF , p is the perimeter of that projection. It turned out if $\nu = 2$, $\sigma = -7/4$, $\delta = 1/2$ then the function

$$V_{\text{elast}} = 0.357469 \frac{8}{15\pi^{1/4}(\theta_A + \theta_B)} \frac{V^2 p^{1/2}}{S^{7/4}},$$

differs from U_{elast} by 0.5% of its value in wide range of the contact ellipse shapes: surely for $b/a \in [0.1, 1]$. Here

$$\theta_\alpha = \frac{1 - \nu_\alpha^2}{\pi E_\alpha}, \quad (\alpha = A, B).$$

Since in the case of the Hertz model the contact spot is the ellipse then the values V, S, p are to be computed by the expressions

$$V = \frac{\pi h^2}{2\sqrt{PQ}}, \quad S = \frac{\pi h}{2\sqrt{PQ}}, \quad p = \frac{4\sqrt{h}(Q/P)^{1/4}E(c_1)}{(PQ)^{1/4}},$$

where the elliptic integral modulus squared this time has the expression $c_1 = 1 - P/Q$. Then taking into account that

$$F_{\text{elast}} = -\frac{\partial U_{\text{elast}}}{\partial h},$$

we get the Vilke formula for the approximate value of the normal force at the contact

$$F_{\text{elast}} = -0.357469 \frac{2}{3(\theta_A + \theta_B)} \frac{\sqrt{E(c_1)}}{P^{3/8}Q^{3/8}} h^{3/2}.$$

Numeric experimental verification showed an application of the above expression for the normal force indeed causes the relative error near the value 0.5% for the contacting bodies configuration coordinates in compare with "exact" Hertz model over long time of simulation. Anyway to estimate with the proper quality the fatigue processes in machines while the lifecycle simulation it is sufficient enough to have an acceptable approximation for the contact forces.

The Vilke formula is essentially simpler than computations in the Hertz model requiring the solution of the

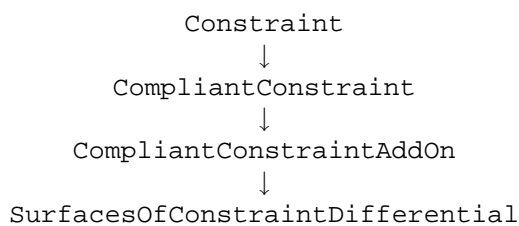
transcendental equation. Volumetric derived algorithm is more reliable than the Hertz one though sometimes due to the differential techniques arranged for the elliptic integrals the Hertz algorithm works even faster than one of Vilke.

4 Implementation

The procedures described above to compute the normal force of an elastic interaction were implemented on Modelica in frame of general approach to construct the objects of mechanical constraint [9]. Strictly speaking in case of the compliant connection the constraint itself is absent. Instead we have an elastic compliance implementing the Hertz contact model. Though the general architecture of the objects interaction conserves completely. Thus for future purpose retain the term “constraint”.

When implementing a class of the compliant interaction it turned out to be useful to split its base classes in two different lines of inheritance: (a) the first one contains mainly the geometric properties, (b) the second line is responsible for the normal force calculation. Thus in the last derived class we use the multiple inheritance allowed in Modelica. An example of the classes hierarchy in the case under consideration see in Figure 3.

The example is one of the simplest ones to test an implementation of the Hertz model: the contact of the ellipsoid and the plane. The left line of inheritance, see Figure 3, concerning mainly with the contact geometric properties



has a common use and doesn't depend on the type of the contacting surfaces. The variables which do depend on such that gradients and the Hesse matrices are evaluated in the class

EllipsoidAndHorizontalPlaneDifferential.

The class SurfacesOfConstraintDifferential is here the most essential derived one. It is responsible for the points P_A and P_B permanent tracking, implements the DAE system (2), (3), and has the following Modelica code

```

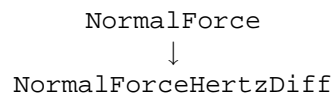
partial model
  SurfacesOfConstraintDifferential
  
```

```

extends CompliantConstraintAddOn;
SI.Velocity[3] drA;
SI.Velocity[3] drB;
ConstraintDetectorRate dmu;
Real lambda;
LambdaRate dlambda;
GradientRate[3] dgradgA;
GradientRate[3] dgradgB;
Real Active(start = 1);
Hessian[3, 3] HessgA;
Hessian[3, 3] HessgB;
equation
der(Active) = 0;
der(rA) = Active*drA;
der(rB) = Active*drB;
der(lambda) = Active*dlambda;
der(mu) = Active*dmu;
dgradgA = cross(InPortA.omega, gradgA)
+ HessgA*(drA - vrA);
dgradgB = cross(InPortB.omega, gradgB)
+ HessgB*(drB - vrB);
dgradgA = lambda*dgradgB +
dlambda*gradgB;
drA - drB = mu*dgradgB + dmu*gradgB;
0 = gradgA*(drA - vrA);
0 = gradgB*(drB - vrB);
HessgA = InPortA.T*HessfA*
transpose(InPortA.T);
HessgB = InPortB.T*HessfB*
transpose(InPortB.T);
end SurfacesOfConstraintDifferential;
  
```

where the variables correspond to ones in (2), (3) in an evident way by use of their names.

In the line of the force properties inheritance



the class NormalForce plays a role of the base class for any implementation of the normal force. In the class NormalForceHertzDiff the normal force besides the elastic Hertzian term has the term of viscosity of the form

$$F_{\text{visc}} = -d(h)\dot{h},$$

where h is the mutual approach. This latter term supposed to arise due to the plasticity properties of the material the bodies made of. It is fair natural to consider the coefficient at \dot{h} to depend upon h [10] since as the mutual approach increases from zero then the contact spot area also increases from zero. Therefore it is quite natural for the plastic resistance to increase continuously from zero.

The class NormalForceHertzDiff Modelica code is long enough thus let us highlight some of its main features, namely the implementation of the auxiliary

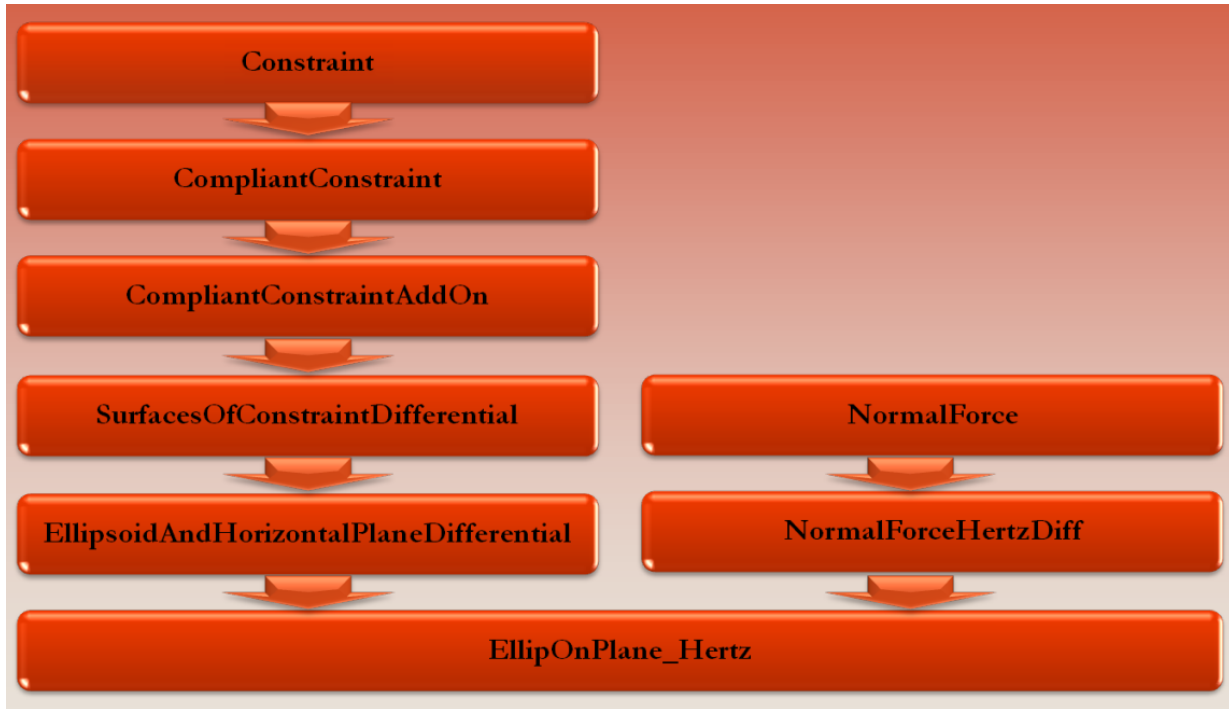


Figure 3: Example of Compliant Constraint Classes Hierarchy

local base $\{\alpha, \beta, \gamma\}$ tracking and equations to compute the solution of the system of the transcendental equations

```

model NormalForceHertzDiff
...
initial equation
  K = CompleteEllofFirstKind(k2);
  E = CompleteEllofSecondKind(k2);
  0.5*CompleteEllofFirstKind(k2)/
    dKdk2(k2) - (1 - k2) = C;
equation
  dgamma = (dgradgA1 -
    normA1*(normA1*dgradgA1))
    /sqrt(gradgA1*gradgA1);
  der(gamma) = dgamma;
  OmegaA = cross(gamma, dgamma);
  der(alpha) = cross(OmegaA, alpha);
  beta = cross(gamma, alpha);
  ...
  der(k2) = dk2;
  dK = if k2 < Accuracy then dKdk2(k2)
    else 0.5*(E - (1 - k2)*K)/k2
    /(1 - k2);
  dE = if k2 < Accuracy then dEdk2(k2)
    else 0.5*(E - K)/k2;
  der(K) = dK*dk2;
  der(E) = dE*dk2;
  C = A1/B1;
  dC = der(C);
  ddK = if k2 < Accuracy then
    d2Kdk22(k2) else 0.25*((1 - k2)*
    (2 - 3*k2)*K - (2 - 4*k2)*E)/k2^2

```

```

    /(1 - k2)^2;
    (3*dK^2 - K*ddK)*dk2 = 2*dK^2*dC;
    ...
end NormalForceHertzDiff;

```

where the variables $K, E, k_2, dK, dE, ddK, A_1, B_1, C$ stand correspondingly for the values $K(c), E(c), c, dK/dc, dE/dc, d^2K/dc^2, P, Q, C$ from previous section. The functions $dKdk_2(k_2), dEdk_2(k_2), d2Kdk_2^2(k_2)$ are used if the modulus is small enough, i. e. regular expressions become inoperative. These functions are computed via expansions of series with the fast convergence for the small modulus. Section of initial equations is needed to initialize a state variables being computed using known expansions for the complete elliptic integrals. These expansions work only once when starting the simulation.

A tangent force at the contact in our case is computed in the class `CompliantConstraintAddOn` and for the simplicity is implemented as a regularized model of the Coulomb friction [6]. Obviously, one can create here even far more complicated models for the tangent force at the contact.

5 Example of the Ball Bearing

The ball bearing model is built up using the architectural principle mentioned above. On the Icon-level of its representation it looks exactly like the model

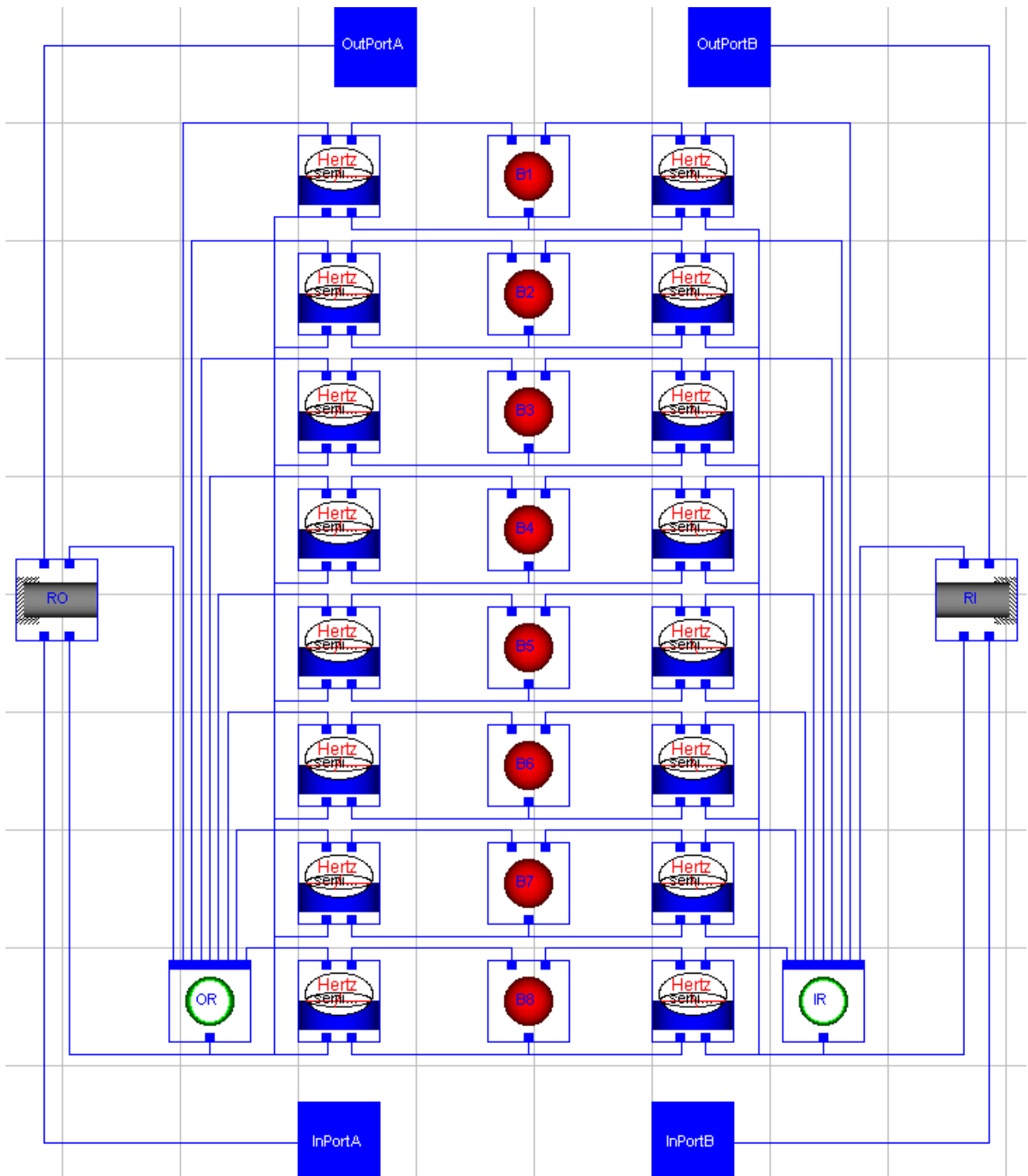


Figure 4: Visual Model of the Ball Bearing

of any constraint: it has two ports of the connector type `KinematicPort` to import the twists of the rigid bodies connected by the bearing, and two ports of the `WrenchPort` type to export the wrenches in directions of the bodies mentioned. Visual model of the ball bearing see in Figure 4

For definiteness the bearing was equipped by eight balls. Each ball has two elastic contacts: one with the inner ring, and one with the outer one. In both cases when contacting the ball simultaneously rolls over the surfaces of the toroidal tubes corresponding to the raceways of the inner and outer rings.

Describe in brief the specifications of the contact between the ball and one of the toroidal raceways. The ring always supposed to be denoted as a body A in the contact object of the ball bearing model, while the ball always denoted as B . All we need to complete the constraint specifications is to define the functions f_A , f_B . In our case we have

$$\begin{aligned} f_A(x, y, z) &= 4R_A^2 (x^2 + y^2) - \\ &\quad (x^2 + y^2 + z^2 + R_A^2 - r_A^2)^2, \\ f_B(x, y, z) &= x^2 + y^2 + z^2 - R_B^2, \end{aligned}$$

where r_A is the toroidal pipe radius, R_A is the radius of the circle being an axis of that toroidal pipe, R_B is the ball radius.

Paying an attention to the ball bearing visual model, Figure 4, note the central column represents eight objects `B1`, `B2`, ..., `B8` of elastic balls. Left and right columns of objects composed by the contact objects between the balls and the outer ring (left column) and inner ring (right column), all implemented using the Hertz model described above. The objects representing in the model the inner and outer rings have the names `IR` and `OR` correspondingly. At left and right extreme sides of the class the objects of rigid constraints are located. These constraints connect the outer and the inner rings objects with the objects of the bodies, outer and inner shafts in our case, attached one with another by the bearing. In the example under consideration the body connected with the outer ring rests w. r. t. AF while the body connected to the inner ring rotates uniformly about z -axis of AF both thus performing the prescribed motion, see the animation image in Figure 5.

The visual model of the example testbench see in Figure 6. To verify the quality of the Hertz model implementation we compared the vectors $\boldsymbol{\gamma}$ and \mathbf{n}_A as functions of time. The computational experiments showed that their coordinates coincide with a very high accuracy. At last yet another remark: to make the simulation even more faster, at least twice, one can apply the

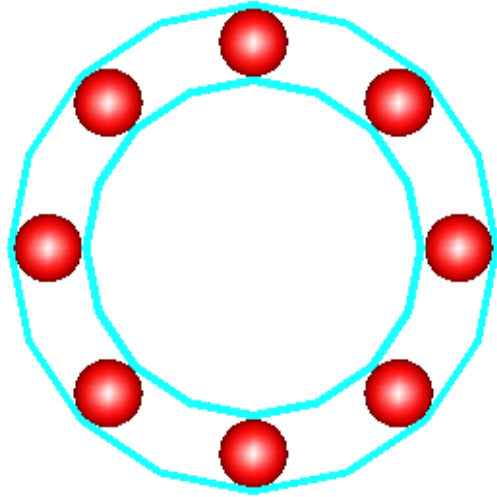


Figure 5: Animation of the Model

simplified expression of the form

$$F_{\text{elast}} = -eh^{3/2},$$

with the constant coefficient e for the normal elastic force at the contact [11]. But it is possible only if the geometric properties (curvatures etc.) don't change while simulating the model. Moreover, for different cases of contacting the coefficient e would have different values. Then its value can be computed using the numerical experiment, or even better using the natural physical experiment. If the motion under simulation is perturbed from its pure case with the constant e then immediately its value begins change in time.

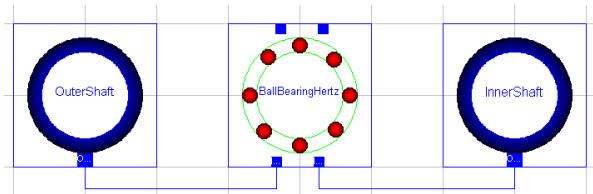


Figure 6: Visual Model of the Testbench

6 Conclusions

Summarizing the results presented above we can split them to the several main remarks influencing the potential directions of future work:

(1) According to an experience accumulated while developing the models simulating the multibody dynamics one can resume the usefulness of the approach when the differential formulations proper applied are preferable in several aspects. It is a real way to han-

dle the transcendental equations in frame of dynamical problems using the ODEs derived from the transcendental ones in combination with the linear solver w. r. t. derivatives of the new state variables.

(2) In particular it turned out an introduction of the component of the ODEs system for the elastic bodies outer surfaces tracking for the contact problem conserves an accuracy and simultaneously improves the reliability of the models. To implement the tracking in case of the complex shape surfaces we have to rearrange only one derived class at the end of the inheritance chain to define an equations for the gradients and Hessians of the surfaces A and B w. r. t. LF s of the bodies. The surfaces supposed smooth enough and without sharp edges but can be described by if -constructs properly arranged.

(3) Implementation of the complete elliptic integrals using ODEs subsystem also was useful: the models became more reliable and faster. For instance, the Hertz algorithm improved as described above turned out to be even faster than the V. G. Vilke one in case of the almost circular contact area.

(4) The algorithm of V. G. Vilke is more reliable and suitable for wide range of the contact area eccentricities simultaneously providing an accuracy of 0.5% with respect to the Hertz-point algorithm.

Regarding the directions of the future work it is evident enough an interest to apply the developed models to different types of appliances with the rotary motions, or to the problems essentially including the effects of friction when contacting.

7 Acknowledgement

The paper was prepared with partial support of Russian Foundation for Basic Research, projects 05-01-00308-a, 05-08-65470, 05-01-00454, SS-6667.2006.1.

References

- [1] Gonthier Y., Lange C., McPhee J., On Implementing a Bristle Friction Model in a Contact Model Based on Volumetric Properties // *Multibody Dynamics 2007, ECCOMAS Thematic Conference, Proceedings, Politecnico di Milano, Milano, Italy, June 25–28, 2007.*
- [2] Hertz H. Über die Berührung fester elastischer Körper // *J. reine und angewandte Mathematik. B. 92. 1882. S. 156–171.*
- [3] Hippmann G., An Algorithm for Compliant Contact Between Complexly Shaped Bodies // *Multibody System Dynamics, 2004, Vol. 12, pp. 345–362.*
- [4] Otter M., Elmqvist H., Lopez J. D., Collision Handling for the Modelica MultiBody Library // *Proceedings of the 4th International Modelica Conference, Hamburg University of Technology, Hamburg–Harburg, Germany, March 7–8, 2005, pp. 45–53.*
- [5] Vilke V. G., On Non-Hertzian Contact of Wheel and Rail // *Research on Problems of Stability and Stabilization of Motion. Reports of the Computing Center of the Russian Ac. of Sc., (in press).*
- [6] Kossenko I. I., Implementation of Unilateral Multibody Dynamics on Modelica // *Proceedings of the 4th International Modelica Conference, Hamburg University of Technology, Hamburg–Harburg, Germany, March 7–8, 2005, pp. 13–23.*
- [7] Landau L. D., Lifshitz E. M., *Theory of Elasticity. 3rd Edition. Landau and Lifshitz Course of Theoretical Physics. Volume 7. — Reed Educational and Professional Publishing Ltd.: Oxford – Boston – Johannesburg – Melbourne – New Delhi – Singapore, 1999.*
- [8] Whittaker E. T., Watson G. N., *A Course of Modern Analysis, — Cambridge University Press: Cambridge – New York – Melbourne – Madrid – Cape Town, 2002.*
- [9] Kosenko I. I., Loginova M. S., Obraztsov Ya. P., Stavrovskaya M. S., *Multibody Systems Dynamics: Modelica Implementation and Bond Graph Representation // Proceedings of the 5th International Modelica Conference, arsenal research, Vienna, Austria, September 4–5, 2006, pp. 213–223.*
- [10] Wensing J. A., *On the Dynamics of Ball Bearings. PhD Thesis. — University of Twente: Enschede, The Netherlands, 1998.*
- [11] Lee S., Park T., Park J., Yoon J., Jeon Y., Jung S., *Fatigue Life Prediction of Guideway Vehicle Components // Multibody Dynamics 2007, ECCOMAS Thematic Conference, Proceedings, Politecnico di Milano, Milano, Italy, June 25–28, 2007.*

Session 2d

Electric Systems & Applications

Modelica'2008

Modelling of Electric Drives using freeFOClib

Dietmar Winkler Clemens Gühmann
Technische Universität Berlin

Chair of Electronic Measurement and Diagnostic Technology
{Dietmar.Winkler|Clemens.Guehmann}@TU-Berlin.de

Abstract

The freeFOClib (short for “free Field-Oriented Control library”) provides a framework for simulations of electric drives with different application purposes. The library can be used to simply build a field-oriented control system for existing machine models from the *Modelica Standard Library*, investigate the impact of electric faults (battery faults, inverter faults, machine faults) on a electric drive system, and run simulations to estimate the fuel consumption of hybrid electric vehicles. The library structure and some of its main components are presented. Simulation results of an electric fault are given as an application example of this library. The freeFOClib will be publicly available in Spring 2008.

Keywords: Modelica, free library, electric machines, field-oriented control, fault simulation, hybrid electric vehicle

1 Introduction

In automotive applications the number of electric motors used is increasing rapidly. Most of them are doing their work without us – the car owners/users – actively noticing it. When a power window is still a quite obvious application for an electric motor, the active controlled throttle valve might not be. And with more and more tasks going to be performed *by-wire* (e.g., braking, steering) the number of electric motors used is due to increase even more. But not only *small* electric motors are present. With the electric motor being used for active propulsion in hybrid electric vehicles (HEV) also the power rating of motors used grows bigger.

But how do all these little and large motors work together? How should the manufacturer develop the controller? What happens if there is a fault in the system?

Will the faults cause serious damage or just minor inconveniences?

All these question could be answered by using simulations to investigate the normal and faulty behaviour. For the creation of such simulation models we need both, appropriate machine models and the suitable machine controllers. The overall simulation model will contain signals from different physical domains (i.e., electrical and mechanical).

The modelling language Modelica¹ was especially developed to simplify the simulation in different physical domains in one simulation model. The multi-domain capability allows us to build simulation models of hybrid electric vehicles easier than than with other simulation tools.

So far our Chair of Electronic Measurement and Diagnostic Technology has investigated different aspects of Modelica with real-time applications and hybrid electric vehicles ([1, 2]).

2 Purpose of the library

The Modelica language is specified in the so called Modelica Specifications [3] and comes with the free *Modelica Standard Library* (MSL) [4] which contains a huge collection of models for different physical domains (e.g., electrical, mechanical, thermodynamical). For the simulation of machines the *Modelica Standard Library* contains a sub-library called Modelica.Electrical.Machines [5]. This library contains basic three-phase models of asynchronous and synchronous machines as well as DC machine models. To control these machines the modeller still has to provide his own controllers since currently there is no *free* Modelica library available to provide complete electric drive models.

¹Modelica® is a free modelling language developed by the Modelica Association → www.modelica.org

So in order to simulate more complex electric drive applications a new “free Field-Oriented Control Library” (freeFOClib[6]) is being developed.

3 Library structure

The freeFOClib should allow the user to model and simulate all aspects of the an electric drive. A standard electric drive normally consist of components like power sources, power electronics, controllers, electric machines, and interfaces. The communication of the blocks can be done either via the classic approach by use of input and output connectors or by the use of bus signals. The bus structure orients itself on the new bus structure of the Vehicle Interfaces Library [7]. This should allow easier simulation of power-train simulations of hybrid electric vehicles, for example.



Figure 1: Top-level packages of the freeFOClib

Figure 1 shows a graphical representation of the uppermost hierarchy level of the library.

The library consists of:

UsersGuide Every Modelica library should contain this. It gives the user information on how to use the library as well as some information about the release history and participating developers.

Examples To get the user going some example simulation models are included. There are sub-packages of examples for the different parts (e.g., complete drive systems, batteries, inverters, machines)

Batteries This is a sub-package that contains different battery models.

Components Models which are library-wide used and therefore do not fit exclusively into any of the other sub-packages are placed here.

Controllers In here controllers for the control of electric drives together with the necessary flux models are placed.

Functions Custom functions which are used by the freeFOClib.

Icons Special icons for the models.

Interfaces The Interfaces sub-package includes different interface models. Mostly they are made of a partial type so that one can simply extend from the interface model which fits the application most.

Inverters These models are used to transform the control signals into electrical signals which can then be applied to the machine models.

Machines This library contains models of synchronous and asynchronous machines of different types.

In addition to the different controller types for the field-oriented control as well as the battery models (important for automotive applications) also new machine models for fault-simulations have been developed. These are using the m -phase presentation where the faults can be introduced directly into the components without the need of a $d - q - 0$ -transformation (see also [8]).

4 Library contents

After a first quick overview of the library we like to explain some the library’s content in more detail.

4.1 Batteries

In the current version of the freeFOClib there are two different types of batteries present. One very simple model consisting of an internal resistor and a controlled signal voltage only. As this is normally not sufficient for more advanced simulations (e.g., simulations of driving cycles of hybrid electric vehicles) an advanced battery model was added. This model is based on models from the *Advisor 2002 Simulink*[®] model. It contains three main sub models: SOC, VocRint, and BatteryECU (see Figure 2).

The advanced battery model includes an active energy management. The state of charge is calculated by SOC. The simple internal resistor of the ideal resistor is replaced by a variable resistor. The value of resistance is

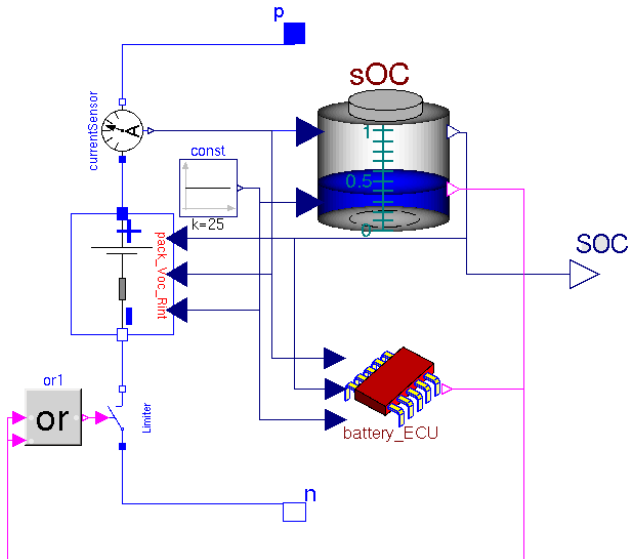


Figure 2: Advanced battery model

controlled by look-up tables and depends on the charging/discharging current, the state of charge, and the temperature. Currently the characteristic behaviour of a Nickel-Metal-Hydrid battery type is implemented. Also included is a switch which acts as a battery protection system and is controlled by the BatteryECU and the SOC models.

4.2 Controllers

This package provides models for the regulation of the control variables flux, speed, torque, and current. There is a series of different control strategies. Which of these strategies is the best applicable depends on the application. In field-oriented control the used flux-model represents a crucial part of the controller. Depending on the machine and the operational range of speed the correct flux model has to be chosen.

The freeFOClib offers a small variety of flux-models (see, for example, [9, 10] for more information on control of electric drives):

- Current models ($I - \vartheta$, $I - \omega$)
- Current-voltage models (UI , $UI - \omega$)
- Voltage models

4.3 Inverters

This sub-package contains different inverter models. For a start there is an IdealInverter which can be used if effects of the power electronic circuitry are not

of interest. This gives the ideal voltage and current signals and does not need much computing power.

When effects of the power electronic are of interest than there are two different types of voltage controlled inverters available:

Space Phasor modulation Depending on the control signal an appropriate voltage space phasor is calculated. The position of the space phasor can then be transformed into firing signals for an m -phase inverter bridge (where m could be any number of phases ≥ 3).

Sinus-Delta modulation Here a sin-wave signal is compared with a high-frequency triangular wave. A logic circuitry then generates the firing signals for the m -phase inverter bridge.

Both non-ideal inverter types have the drawback that they need a lot of computational power. This has caused by the need to restart the numerical solver whenever an event (e.g., switch from one inverter leg to the other) is triggered.

4.4 Machines

Fortunately, if a modeller tries to build a standard electric drive with standard 3-phase machines he can just pick them out of the Modelica.Electrical.Machines library. These standard machines can be used in combination with our controllers and inverters.

Unfortunately, there is a restriction on the number of phases (i.e. $m = 3$). The freeFOClib provides asynchronous and synchronous induction machine models for m -phases and different architectures. The sub-package for synchronous machines contains m -phase models with electrical excited and permanent magnet rotors. The sub-package for asynchronous machines contains m -phase models with squirrel-cage and slip-ring rotor. In each of the models the stator inductance can be changed during simulation time to investigate fault impacts.

In contrast to the machine models from the *Modelica Standard Library* the m -phase models of the freeFOClib are not modelled in the so called $d - q - 0$ -frame but in the m -phase system. This might seem odd since the $d - q - 0$ -frame was actually introduced to reduce the computational demand of machine simulations. However when developing the freeFOClib one of the requirements was to be able to simulate faults and therefore unsymmetrical systems. As soon

as an electrical m -phase system becomes unsymmetrical all the computational benefit of the $d - q - 0$ transformation is lost. More about the underlying theory of the machine models can be found here [8].

5 Library applications

The development of the `freeFOClib` was started with specific purposes in mind:

- field-oriented control of induction machines
- fault-simulations to investigate electrical and mechanical impacts of machine faults
- state of charge estimations for batteries in HEV applications
- investigate adaptive controller algorithms for electric machines

In automotive applications, for example, often the term drivability of a car is used. With drivability the car manufacturers often relate to the overall operating qualities of the power train. This could include things like idle mode characteristics, throttle response, and acceleration capability. In a hybrid electrical vehicle for example we got an electric motor acting directly or indirectly on the drive train.

The `freeFOClib` contains an example model which allows for simulation of three different types of faults, i.e., faults of the battery, faults in the inverter, and faults in the machine (see Figure 4).

5.1 Inverter and battery faults

To simulate faults in the battery and/or in the inverter, the example model in Figure 4 contains a fuse component `fuse_DC`. This model disconnects the DC current when a surge current is detected that violates the maximum rating of the battery. The surge-proof fuse is not triggered right away but after a first order delay time which can be parametrised. Whenever the fuse is triggered the inverter gets a signal which will switch off the firing signals for the inverter bridge in turn.

Another kind of battery fault would be a short circuit of the supply side of the inverter. This is accomplished by a simple switch that is triggered by a boolean signal and which connects both support voltage connectors of the inverter.

And at last a fault of firing signals can be applied directly via a signal `inverter_fault`. The combination of the different switches gives the ability to build even more fault scenarios.

5.2 Machine faults

In the electric machine models of our library the following fault scenarios can be simulated:

- open-circuit of a stator phase (e.g., a connecting cable is broken)
- short-circuit phase to ground (e.g., insulation failure because of mechanical damage)
- short-circuit of one or more phase windings (e.g., insulation failure because of thermal stress within the stator or rotor)

Each of these faults will have some influence of the torque produced by the electric drive.

5.2.1 Short-circuit phase to ground

In Figure 3 you can see the simulation results of a synchronous machine with an electric excited rotor. At the time of $T = 2\text{sec}$ one stator phase is connected to ground. The figure shows all three phase currents and the mechanical torque over time.

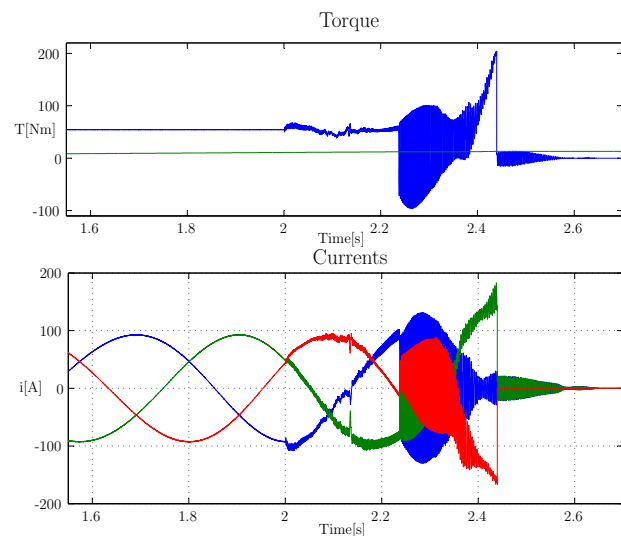


Figure 3: Fault scenario: short-circuit of one stator phase to ground

At the beginning the electric machine runs at a constant speed and with a constant load torque applied to the shaft end. When the short-circuit occurs the controller tries to keep the torque at a constant level but can only do so for a certain amount of time until the fuse finally gets triggered because of over-current. With such kind of simulation model one could for example try to find the optimum kind of fuse which

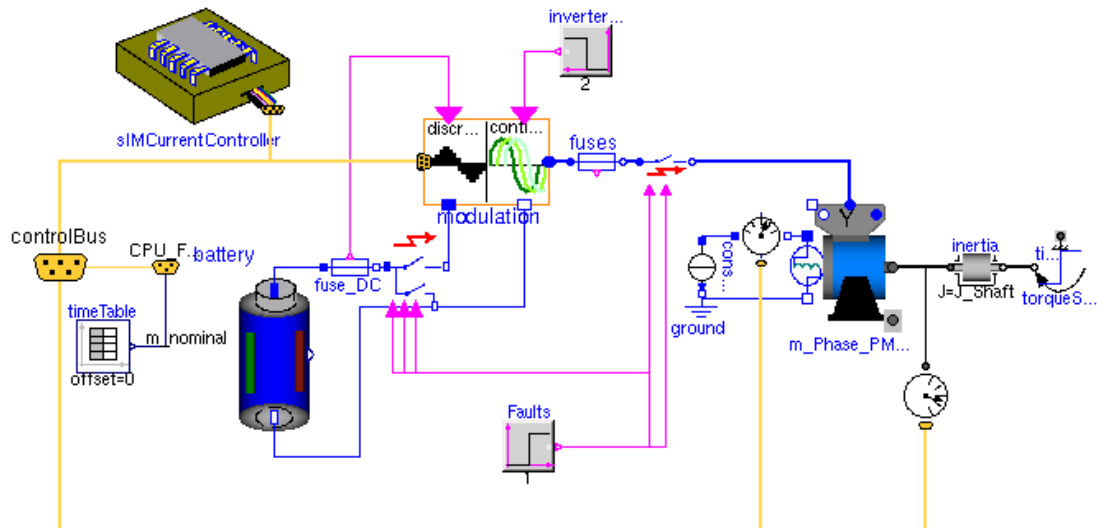


Figure 4: Example model of an electric drive system.

is sluggish enough to withstand short power surges whilst still protecting the drive's power electronic.

5.2.2 Short-circuit within a phase winding

In Figure 5 a fault of the insulation between the phase windings of a stator coil is modelled. Such a fault can be caused by, for example, over-temperature or overload which in turn leads to an overheated stator winding. This behaviour is modelled by reducing the inductance value abruptly by 20 percent. In Figure 5 you can see the three phase currents and the mechanical torque over time just before and after the connection of one phase was opened at the time of $T = 2\text{sec}$.

At first sight the electrical impact seems not to be very drastic. However since the field-oriented control now calculates the wrong control values the torque starts to oscillate quite considerable. If this electric drive is applied in a hybrid electric vehicle, for example, this could lead to reduction of drive comfort. But not only this, depending on the mechanical system such oscillation could become unstable and cause major damage.

5.3 Field-oriented control loop

Having a variety of different flux-models available allows the modeller to investigate different kind of control strategies for electric drives. So for example by varying some of the controllers parameters (e.g., the machine rotor resistance) one can test how robust the drive control behaves at a certain rotor speed when using different flux-models for the estimation of the flux position.

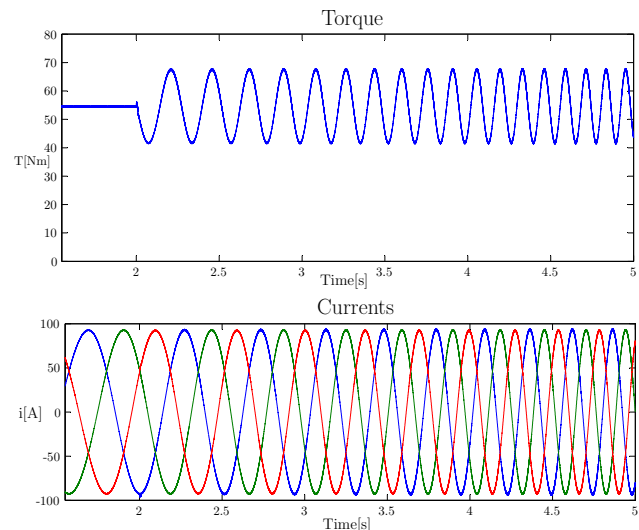


Figure 5: Fault scenario: short-circuit of phase windings

6 Future work

The main task for the future is the extension of the freeFOClib. Here is a short collection of ideas we are currently having:

- further clean up of the structure so that the usage is more intuitive
- add more controller types which allow also the investigation of different control architecture not just pure parameter variations
- enhance battery models with temperature models

(at the moment only look-up tables are used to calculate the resistance)

- verification of machine models
- investigate simplifications to make the machine and inverter models real-time capable

A pure estimate of the simulated values might give some clues on the general behaviour during faults. However to actually use the simulation to gain useful information (e.g., for programming a controller for the power electronics) we need more than just estimates. So the simulation model has to be verified by doing real measurements using a real electric motor. It is planned to set up test-bench system consisting of a asynchronous induction machine of from currently available hybrid electric vehicle and an electric load machine.

The first public official (pre)release is due in spring of 2008. There will be a public development repository available. For any news on the freeFOClib see www.freefoclib.org. On that site a mailing-list is also available to keep you up to date automatically.

7 Acknowledgements

We hereby like to acknowledge that a lot of work on the library was done by students as part of their student research projects:

- Stefan Rinderer, “*Development of a Modelica Library for the Field-Oriented Control of Electric Machines*”, student research project, supervisor: Dietmar Winkler, Chair of Electronic Measurement and Diagnostic Technology, TU Berlin, 16th April 2007
- Eduard Bakhach and Florian Döring, “*Modelling of a Field-Oriented Synchronous Inductionn Machine with Modelica*”, student research project, supervisor: Dietmar Winkler, Chair of Electronic Measurement and Diagnostic Technology, TU Berlin, 31st August 2007

References

- [1] D. Winkler and C. Gühmann, “Synchronising a Modelica Real-Time Simulation Model with a Highly Dynamic Engine Test-Bench System,” in *5th International Modelica Conference* (D. C. Kral and A. Haumer, eds.), vol. 1, (Vienna), pp. 275–281, Modelica Association, arsenal research, 4-5 Sep 2006.
- [2] D. Winkler and C. Gühmann, “Hardware-in-the-Loop simulation of a hybrid electric vehicle using Modelica/Dymola,” in *The 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium & Exposition* (E. . Secretariat, ed.), (Yokohama, Japan), pp. 1054–1063, EVS, Japan Automobile Research Institute, 23-28 October 2006.
- [3] Modelica Association, *Modelica[®] – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification*, version 3.0 ed., 5th September 2007.
- [4] Modelica Association, *Modelica[®] - Free library from the Modelica Association*, version 3.0 ed., 2008.
- [5] C. Kral and A. Haumer, “Modelica libraries for dc machines, three phase and polyphase machines,” in *Proceedings of the 4th International Modelica Conference* (G. Schmitz, ed.), pp. 549–558, Modelica Association, March 7-8 2005.
- [6] D. Winkler, E. Bakhach, F. Döring, and S. Rinderer, “freeFOClib - A free Field-Oriented Control library for Modelica.” unreleased, see www.freefoclib.org for any news. Due in, Spring 2008.
- [7] M. Dempsey, M. Gäfert, P. Harman, C. Kral, M. Otter, and P. Treffinger, “Coordinated Automotive Libraries for Vehicle System Modelling,” in *Proceedings of the 5th International Modelica Conference* (D. C. Kral and A. Haumer, eds.), vol. 1, (Vienna), pp. 33–41, Modelica Association, arsenal research, September 2006.
- [8] D. Winkler and C. Gühmann, “Modelling of Electrical Faults Using Modelica,” in *Proceedings of 'The 48th Scandinavian Conference on Simulation and Modeling (SIMS 2007)*, Scandinavian Simulation Society, 30th - 31st October 2007.
- [9] W. Leonhard, *Control of Electrical Drives*. Springer, 3rd ed., 2001.
- [10] D. Novotny and T. Lipo, *Vector Control and Dynamics of AC Drives*. Clarendon Press Oxford, 1996.

Electromagnetic Actuator Modelling with the Extended Modelica Magnetic Library

Thomas Bödrich

Dresden University of Technology, Institute of Electromechanical and Electronic Design
01062 Dresden, Germany
Thomas.Boedrich@mailbox.tu-dresden.de

Abstract

Recently, an improved and extended version of a Modelica library for lumped network modelling of electromagnetic devices has been released [1]. This library is intended for both rough design of the magnetic circuit of those devices as well as for their system simulation. Improvements compared to a first realisation of this library [2] will be discussed and utilization of the extended library for modelling of electromagnetic actuators will be illustrated. To support the work with this library, focus will be on newly-implemented features and on peculiarities of lumped network modelling of electromagnetic actuators rather than on an introductory explanation of the underlying concept of magnetic flux tubes.

Keywords: lumped magnetic network; electromagnetic/electrodynamic actuator; system design

1 Introduction

The well-established concept of magnetic flux tubes enables the modelling of magnetic fields with lumped networks. In the decades prior to the broad availability of software packages based on finite element techniques, this was the only efficient means for model-based design of the magnetic circuit of electromagnetic devices such as transformers, motors and electromagnetic or electrodynamic actuators. The method of magnetic flux tubes, its utilization for the design of electromagnetic devices as well as derivation of the permeance of many flux tube shapes are explained in-depth for example in [3], [4] and [5] and in condensed form in the documentation of [1].

Even though finite element techniques allow for more accurate calculation of magnetic fields, lumped magnetic network models are still an efficient means for initial rough design of electromagnetic devices and for simulation of their dynamic behaviour during system design. This is due to the relatively little ef-

fort needed for creation of *rough* magnetic network models and to the low computational effort for dynamic simulation compared to finite element techniques.

A first realisation of a Modelica library for lumped network modelling of electromagnetic devices was presented in [2]. This library was improved and extended and is now available [1]. Some of the important improvements of the extended library compared to the prior version are:

- a more general approach for calculation of magnetic reluctance forces (section 2),
- redesigned and extended components for representation of typical flux tube shapes (section 2),
- additional soft magnetic and permanent magnetic materials (section 3),
- more accurate modelling of magnetic leakage fields for *dynamic* simulation by splitting of the magnetomotive force imposed by a coil into separate sources (section 4) and
- additional examples of different modelling depths.

2 Reluctance Force Calculation

Generally, the thrust F developed by a translatory electro-magneto-mechanical actuator (similar for the rotational case with torque and angular position) is equal to the change of magnetic co-energy W_m^* with armature position x according to

$$F = \frac{\partial}{\partial x} \int_{(i)} \Psi di = \frac{\partial}{\partial x} W_m^* \quad (1)$$

(Ψ flux linkage, i actuator current) [4]. In lumped magnetic network models, the above equation simplifies to

$$F = \frac{1}{2} \sum_{i=1}^{n_{linear}} V_{mag i}^2 \frac{dG_{mi}}{dx} \quad (2)$$

where n_{linear} is the number of flux tube elements with constant relative permeability that change their per-

meance G_{m_i} with armature position (index i), V_{mag_i} the magnetic voltage across each respective flux tube and dG_{m_i}/dx the derivative of the respective permeances with respect to armature position. Transition from the general formula based on magnetic co-energy (Eq. (1)) to Eq. (2) is outlined in [4] for the reciprocal of the permeance, i.e. for the magnetic reluctance. Compared to the reluctance force calculation with *Maxwell's* formula used in [2], the newly-implemented approach according to Eq. (2) simplifies force calculation for air gaps different from a simple cylindrical or prismatic shape with axial magnetic flux (see below).

The usability of Eq. (2) is not restricted solely to network models with constant relative permeabilities μ_r of the flux tubes. However, it is required that flux tubes with a dependency of the permeability on the flux density B such as ferromagnetic components with non-linear characteristics $\mu_r(B)$ do not change their shape with armature motion (e.g. portion of a solenoid plunger where the magnetic flux passes through in axial direction). This limitation is not a strong one, since the permeance of non-linear, high permeable ferromagnetic flux tube elements and its change with armature position compared to that of air gap flux tubes can be neglected in most cases. Because of this constraint, the dimensions of possibly non-linear flux tube elements in sub-package `FluxTube.FixedShape` are fixed, whereas the dimension l in direction of motion of the linear flux tube elements in sub-package `FluxTube.Force` can vary during simulation. Elements with fixed shape are intended e.g. for modeling of transformer cores or ferromagnetic sections of actuators. Force elements are to be used for modelling of working air gaps or for moving permanent magnets of actuators.

In order to fulfill Eq. (2) in a magnetic network model of a translatory actuator, the reluctance force F_m of each flux tube with force generation is calculated in the respective element accordingly:

$$F_m = \frac{1}{2} V_{mag}^2 \frac{dG_m}{dx} \quad (3)$$

V_{mag} denotes the magnetic voltage across the flux tube and dG_m/dx is the derivative of flux tubes permeance G_m with respect to armature position x . Summation of all particular reluctance forces to the actuators net force F according to Eq. (2) is induced by connecting the translatory flange connectors of all flux tube elements with force generation in an actuator model (see example in Figure 4b).

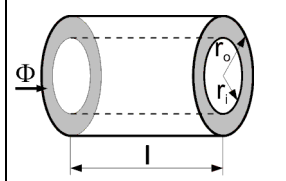
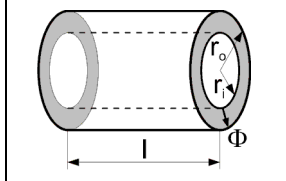
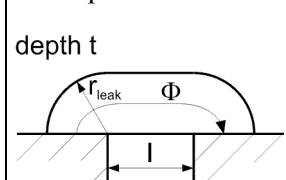
In a particular actuator model, the derivative dG_m/dx of the models flux tubes with force generation and hence the sign of the generated reluctance force de-

pends on the design of the actuators magnetic circuit and on the definition of the armature coordinate x . To cover all possible conditions in a uniform way, the above derivative is calculated as follows for all flux tubes with force generation of sub-package `FluxTube.Force`:

$$\frac{dG_m}{dx} = \frac{dG_m}{dl} \frac{dl}{dx} \quad (4)$$

For the flux tubes defined in this package with their rather simple shapes, the derivative dG_m/dl is given analytically (Table 1). l is the flux tube dimension that changes with armature motion. dl/dx is an integer parameter that must be set to +1 or -1 according to the magnetic circuit and the definition of the armature coordinate x of an actuator. For more complex shapes and variations of dimensions with armature motion, the derivative dG_m/dl must be provided analytically during model development, preferably by extending the partial model `FluxTube.Force.PartialForce`.

Table 1: Selected flux tube elements with reluctance force generation (subset of `FluxTube.Force`), permeance G_m and analytic derivative dG_m/dl

<p>(Hollow) cylinder with axial magnetic flux</p> 	$G_m = \frac{\mu_0 \mu_r A}{l}$ $\frac{dG_m}{dl} = -\frac{\mu_0 \mu_r A}{l^2}$ <p>with $A = \pi(r_o^2 - r_i^2)$</p>
<p>Hollow cylinder with radial magnetic flux</p> 	$G_m = \frac{\mu_0 \mu_r 2 \pi l}{\ln(r_o/r_i)}$ $\frac{dG_m}{dl} = \frac{\mu_0 \mu_r 2 \pi}{\ln(r_o/r_i)}$
<p>Simple leakage flux tube around cylindrical or prismatic poles</p> <p>depth t</p> 	$G_m = \frac{2 \mu_0 t}{\pi} \ln\left(1 + \frac{\pi r_{leak}}{2 l}\right)$ $\frac{dG_m}{dl} = -\frac{\mu_0 t r_{leak}}{l^2 \left(1 + \frac{\pi r_{leak}}{2 l}\right)}$

The leakage flux tube shown in Table 1 provides a simple but efficient means to account for leakage around prismatic or cylindrical poles. In the latter case, depth t is equal to the circumference of a circle given by the average radius of the flux tube. Due to the constant radius r_{leak} of the leakage field, the model is rather simple. In reality, r_{leak} is approximately constant for air gap lengths l greater than this radius, but decreases with air gap lengths less than r_{leak} . This decrease for small air gaps is neglected in the model since the influence of the leakage flux tube compared to that of the enclosed main air gap (connected in parallel, see example in Figure 4b) decreases for decreasing air gap length l .

The sub-package `FluxTube.Leakage` contains flux tube shapes typical for leakage flux around prismatic or cylindrical poles that do not change their shape with armature motion. Hence, the permeance of these flux tubes does not depend on armature position and these elements do not contribute to the thrust of a reluctance actuator.

3 Modelling of Magnetic Materials

3.1 Soft Magnetic Materials

The characteristics of the relative magnetic permeability versus flux density $\mu_r(B)$ of various steels, electric sheets (Figure 1) and high permeable materials are included in `Material.SoftMagnetic`. These characteristics are uniformly approximated with a function adapted from [6]:

$$\mu_r = 1 + \frac{\mu_i - 1 + c_a B_N}{1 + c_b B_N + B_N^n} \quad \text{with} \quad B_N = \left| \frac{B}{B|_{\mu_r=\mu_{\max}}} \right|. \quad (5)$$

This approach assures proper behaviour throughout the complete range of flux density. Overshoot and extrapolation errors especially at high flux densities as possible with spline interpolation or with the Modelica Standard Library's table interpolation with continuous derivative can not occur with properly chosen parameters. Two of the five parameters of Eq. (5) have a physical meaning, namely the initial relative permeability μ_i at $B=0$ and the magnetic flux density at maximum permeability $B|_{\mu_r=\mu_{\max}}$. Addition of new soft magnetic materials requires determination of the function parameters with a non-linear curve fit outside of the library. For the included materials, attention was paid to accurate fits of μ_r for data points at high flux densities ($B > 0.8$ T). This is because of the large influence of saturated ferromagnetic materials on the behaviour

of a device. It must be noted that a measured characteristics $\mu_r(B)$ strongly depends on the shape, machining state and heat treatment of a sample, and on the measurement conditions. Hence, different characteristics are possible for similar materials as Figure 1 indicates.

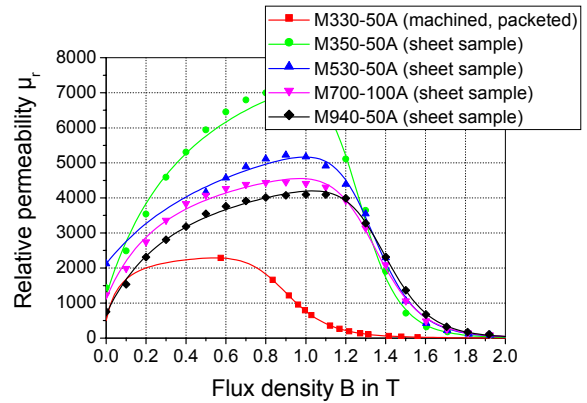


Figure 1: Approximated magnetization characteristics (solid lines) and original data points of included electric sheet materials (all valid for 50 Hz)

Magnetization characteristics that are simulated with Eq. (5) are shown for an electric sheet as an example in the familiar form of flux density vs. field strength $B(H)$ in Figure 2.

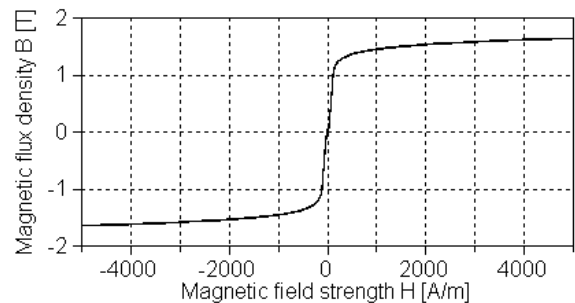


Figure 2: Simulated magnetic flux density vs. magnetic field strength $B(H)$ of electric sheet M350-50A

3.2 Hard Magnetic Materials

For common permanent magnetic materials, typical values for remanence $B_{r\ ref}$ and coercivity $H_{cB\ ref}$ at a reference temperature T_{ref} and the temperature coefficient α_{Br} of remanence are provided in sub-package `Material.HardMagnetic` (Figure 3). Records for additional materials can be defined as needed.

Linear demagnetization curves are modelled. The characteristic, temperature-dependent "knee" of many permanent magnetic materials is not considered, since proper design of permanent magnetic cir-

circuits should avoid operation of permanent magnets "below" that point due to partial demagnetization. As a result, the temperature coefficient of coercivity is not considered. Only the temperature coefficient of remanence α_{Br} is accounted for, since this describes the dependency of the demagnetization curve on the temperature sufficiently for the region "above the knee-point". Remanence B_r and coercivity H_{cB} that are effective at a given operating temperature T are calculated according to

$$B_r = B_{r,ref}(1 + \alpha_{Br}(T - T_{ref})), \quad (6a)$$

$$H_{cB} = H_{cB,ref}(1 + \alpha_{Br}(T - T_{ref})) \quad (6b)$$

in the model `Material.HardMagnetic.PermanentMagnetBehaviour`. Thus, the demagnetization curves shown in Figure 3 are shifted depending on the operating temperature T and the temperature coefficient of remanence α_{Br} . Usage of the above-mentioned component and modelling of a permanent magnet are demonstrated in the library in `Examples.ElectrodynamicalActuator.MagneticCircuitModel` (Figure 9).

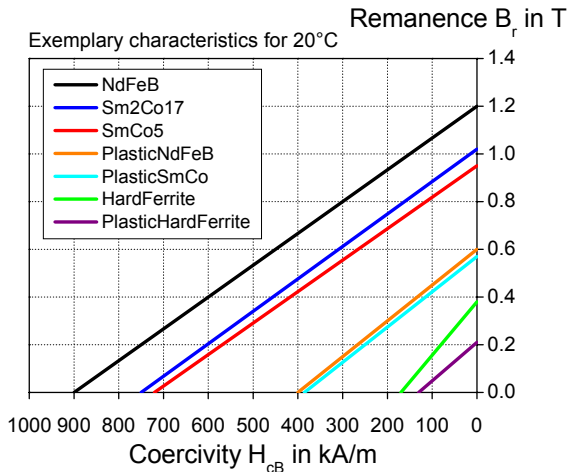


Figure 3: Modelled demagnetization characteristics of included common permanent magnetic materials

4 Modelling of Electromagnetic Actuators

As an example of a reluctance actuator, a simple axisymmetric lifting magnet with planar end planes of armature and pole is modelled in sub-package `Examples.ElectromagneticActuator` of the library. Two lumped magnetic network models of different modelling depth are included. In a `SimpleSolenoidModel`, radial leakage that is typical for tubular reluctance actuators is neglected. Higher

accuracy can be gained from an `AdvancedSolenoidModel` in which the coil-imposed magnetomotive force (mmf) is split into two separate mmf sources and the radial leakage flux between armature and yoke is accounted for with leakage permeance $G_{mLeakRad}$ (Figure 4). This leakage affects the static force-stroke characteristic and the inductance of an actuator and hence its dynamic behaviour especially at large air gaps, as discussed below.

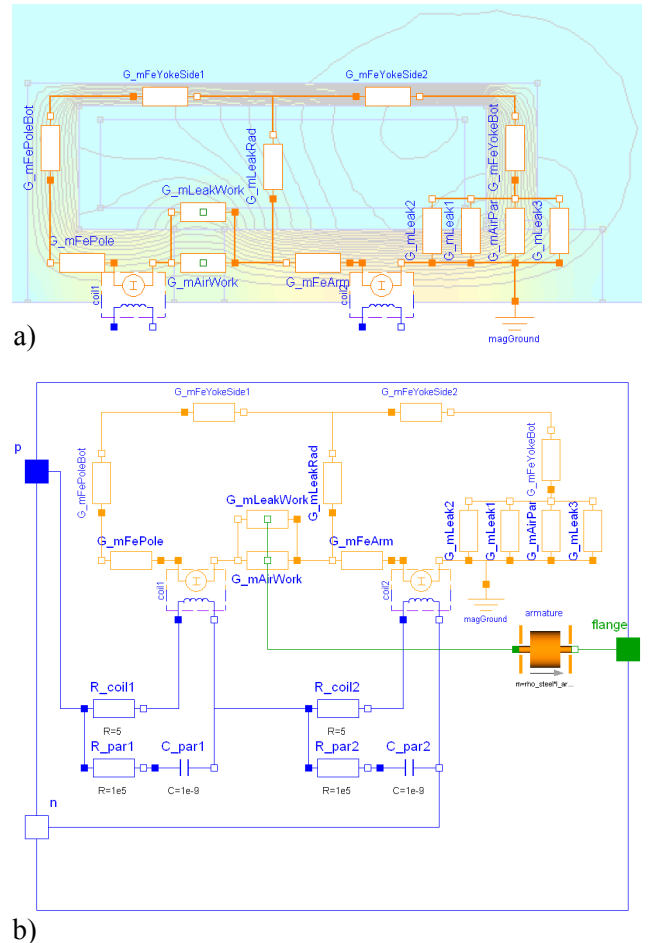


Figure 4: *Magnetic network model of an exemplary solenoid actuator: a) Permeances superimposed on FEA field plot (half-section) for illustration; b) Complete actuator model including electrical subsystem*

Simulation of lumped magnetic network models with multiple mmf sources is easily possible for the stationary case where the coupling between the electrical and the magnetic domain according to *Faraday's law*

$$u = -w \frac{d\Phi}{dt} \quad (7)$$

(u voltage induced in a coil at change of magnetic flux $d\Phi/dt$, w number of turns) needs not to be considered. For dynamic simulation, however, split mmf

sources are challenging due to the strong interactions between the electrical and magnetic domain given by *Faraday's* law (Eq. (7)) and *Ampere's* law

$$V_{mag} = i w \quad (8)$$

(V_{mag} magnetic voltage across mmf source, i current). Eq. (7) and Eq. (8) are implemented in the electromagnetic converters *coil1* and *coil2* of Figure 4b. The parasitic capacitances c_{par1} and c_{par2} are required to ensure definite voltages across both halves of the coil. The values of the auxiliary resistors R_{par1} and R_{par2} have been chosen so that simulation is numerically stable and fast.

In order to evaluate the accuracy in static and dynamic behaviour to be achieved with the above-mentioned lumped network models, a dynamic model of the example actuator based on more accurate finite element analysis (FEA) has been created as a reference (Figure 5). In this model, one of several possibilities to describe an electromagnetic actuator's dynamic behaviour with look-up tables obtained from stationary FEA is implemented. Here, the actuator force $F(x, I)$ and the flux linkage $\psi(x, I)$ were calculated for different fixed armature positions x and stationary currents I with FEA. The derived tabular function $\psi(x, I)$ was inverted to $I(x, \psi)$ as required in the model. The voltage across the current source i_{Coil} is according to *Kirchhoff's* voltage law for the actuator's electrical subsystem, which is given by

$$u = iR_{coil} + \frac{d\Psi(x, i)}{dt} \quad (9)$$

(u voltage across coil terminals p and n , i current, R_{coil} coil resistance).

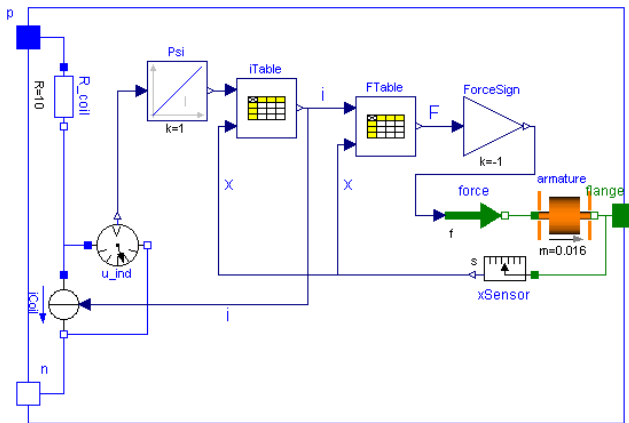


Figure 5: Dynamic model of sample actuator based on look-up tables obtained from stationary FEA

In Figure 6, the accuracy in stationary behaviour to be achieved between the network model of Figure 4, the above-mentioned simple network model without

radial leakage between armature and yoke and the more accurate FEA-based model of Figure 5 is compared. All curves were derived from a quasi-static forced movement of the models armature at a given, constant voltage. The negative sign of the actuator force is due to the definition of the armature coordinate x which is equal to the length of the working air gap.

The static force-stroke characteristics $F(x)|_{i=\text{const}}$ of all three models are similar. However, differences between both network models can be observed for the magnetic flux through the armature and for the static inductance $L_{stat} = \psi/I$, especially at large air gaps where the leakage permeance $G_{mLeakRad}$ is large compared to the net permeance $G_{mAirWork} + G_{mLeakWork}$ of the air gap region. The differences in static inductance between both network models will result in different dynamic behaviour of both models, as Figure 7 indicates.

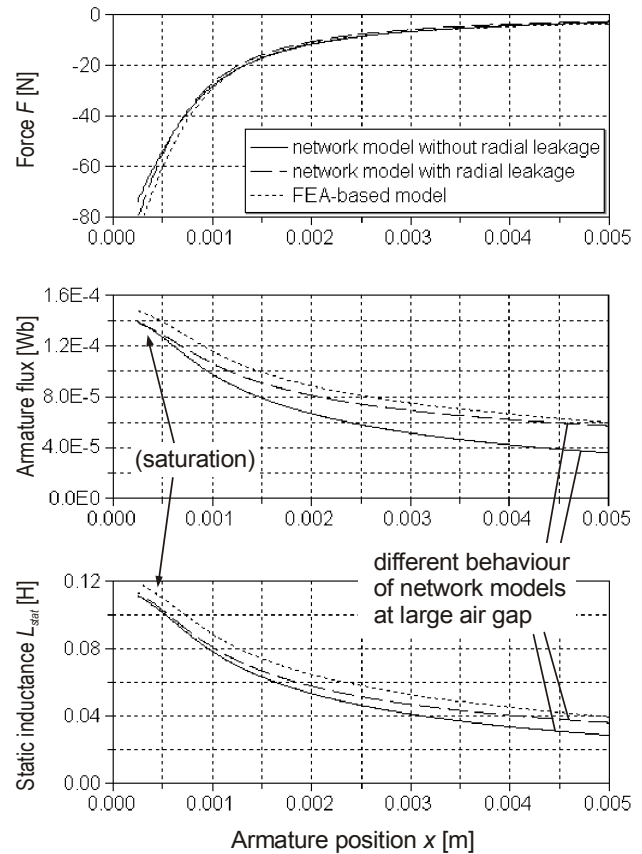


Figure 6: Comparison of stationary behaviour of a simple lumped magnetic network model without radial leakage, the network model of Figure 4b with radial leakage and the FEA-based model of Figure 5

Differences in model behaviour during a simulated pull-in stroke of the armature are shown in Figure 7. At time $t=0$, a voltage step is applied to both lumped network models and to the FEA-based model of the

actuator. The characteristic current drop during pull-in is due to the motion-induced emf and to the increase of the inductance with decreasing air gap. One can see that the simulated current rise of the network model with radial leakage is closer to that of the FEA-based reference model than the current rise of the simple network model without radial leakage. As a result, the magnetic force of latter model shows the fastest rise and simulated armature motion is faster than that of the FEA-based model.

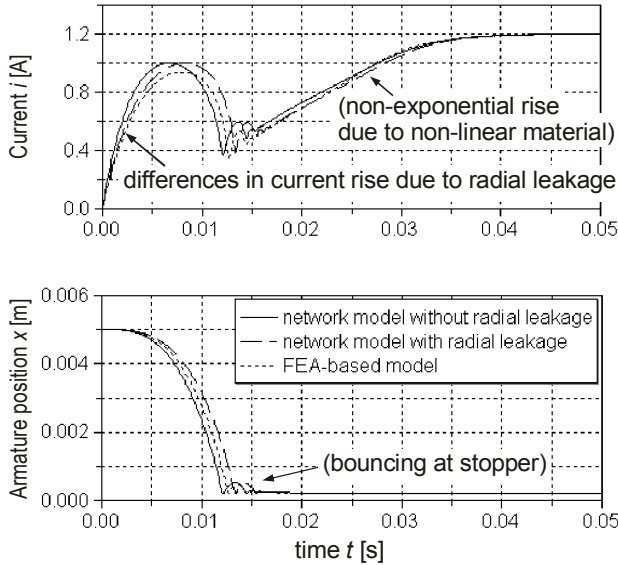


Figure 7: Comparison of the models dynamic behaviour with a simulated pull-in stroke (voltage step applied at time $t = 0$)

5 Modelling of Electrodynamic Actuators

Similar to the well-known behavioural model of a rotational DC-Machine, the electro-mechanical conversion process of translatory electrodynamic actuators (either moving coil or moving magnet type) can be described with a motor constant c_m :

$$F_L = c_m i, \tag{10a}$$

$$u_i = c_m v. \tag{10b}$$

F_L denotes the electrodynamic or *Lorentz* force, i the current, u_i the induced back-emf and v the velocity of the armature. During design of such actuators, the motor constant as well as the motor inductance can be determined by means of a lumped magnetic network model of an actuator’s magnetic circuit.

As an example, Figure 8a shows the principal structure of an axisymmetric translatory electrodynamic actuator with moving coil. The flux lines and the flux

density of the permanent magnetic field were calculated with FEA as a reference (Figure 8b, half-section). Figure 9 shows a Modelica model of that actuator intended for dimensioning of the actuator’s magnetic circuit and winding and for subsequent dynamic simulation at the system level. This example is included as `MagneticCircuitModel` in package `Examples.ElectrodynamicActuator` of the library.

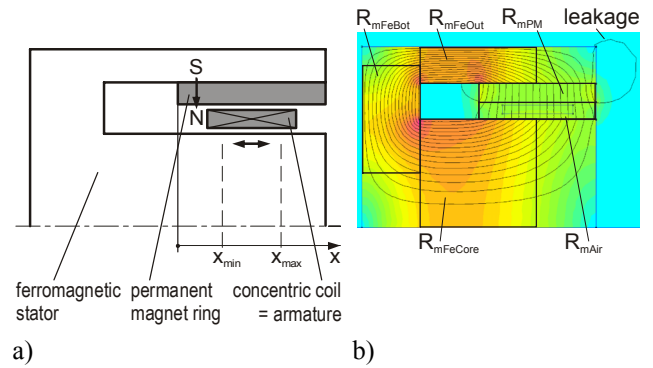


Figure 8: *Translatory electrodynamic actuator*: a) Structure; b) FEA field plot of permanent magnetic field and partitioning into flux tubes

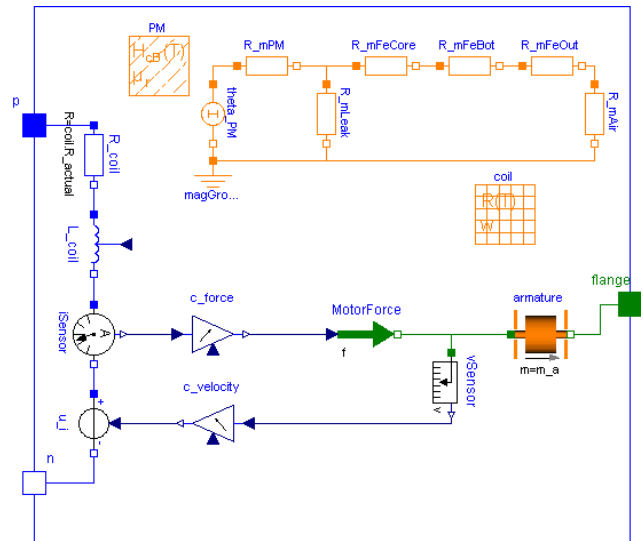


Figure 9: Dynamic model of the electrodynamic actuator of Figure 8 with lumped magnetic network for determination of motor constant and inductance

Although the formula for estimation of the total magnetic reluctance $R_{m\ tot}$ perceived by the coil is rather simple in this model, comparison with FEA showed that it is well-suited for initial rough design and system simulation of the actuator. The relative difference of the inductance

$$L_{coil} = w^2 / R_{m\ tot} \tag{11}$$

(w number of turns) compared to more accurate FEA is -12% for the armature in mid-position. For useful operating currents, the relative differences of the air gap flux density and the resulting *Lorentz* force to the values obtained with FEA are within -5% to -8%. This accuracy is sufficient for both initial actuator design and system simulation.

6 Summary and Outlook

New features and improvements of a Modelica library for lumped network modelling of electromagnetic devices [1] were presented. Two examples, an electromagnetic and an electrodynamic translatory actuator showed that magnetic network models can be used efficiently for the rough design of such devices. These examples are included in the respective sub-packages of the library and can thus be examined in depth.

Despite the simplicity of the presented models, their accuracy is sufficient for preliminary design, as comparisons with more accurate FEA revealed. The dimensions, cross-sections and winding parameters found with magnetic network models can be input data to a subsequent detailed design with FEA or similar techniques, if necessary.

A suitable approach for lumped network modelling of leakage fields for *dynamic* simulation was discussed in detail. Insertion of appropriate leakage permeances into a magnetic network model requires splitting of a devices coil into separate mmf sources. This is challenging due to the multiple couplings between the electrical and the magnetic domain in such a model.

For some electromagnetic devices, lumped network modelling is not possible or reasonable due to distinct leakage fields, complex pole shapes or - with certain actuators - flux tubes and network structures that vary considerably with armature motion. Examples of such devices are inductors without a closed core or proportional solenoids. In this case, dynamic models for system simulation can be created with look-up tables obtained from stationary FEA or similar techniques. One of several possible structures of such a model is shown in section 4.

At present, network modelling of translatory actuators is supported by the library. If needed, the provided model components can be adapted to network modelling of rotational devices. Hysteresis of ferromagnetic materials is currently neglected, since the intention of the library is to support the rough design of electromagnetic devices where a limited accuracy is often sufficient. If necessary, the provided flux

tube elements can be extended so that hysteresis is considered.

It is planned to include the developed library into the Modelica Standard Library after an evaluation period.

References

- [1] Bödrich, T.: Modelica_Magnetic library. <http://www.modelica.org/libraries/> (Oct 11, 2007)
- [2] Bödrich, T.; Roschke, T.: A Magnetic Library for Modelica. Proc. of 4th International Modelica Conference, Hamburg, March 7-8, 2005, pp. 559-565
- [3] Roters, H.: Electromagnetic Devices. New York: John Wiley & Sons 1941
- [4] Kallenbach, E.; Eick, R.; Quendt, P.; Ströhla, T.; Feindt, K.; Kallenbach, M.: Elektromagnete: Grundlagen, Berechnung, Entwurf und Anwendung. 2nd ed. Wiesbaden: B.G. Teubner 2003
- [5] Hendershot, J.R. Jr.; Miller, T.J.E.: Design of Brushless Permanent-Magnet Motors. Magna Physics Publishing and Oxford University Press 1994
- [6] Roschke, T.: Entwurf geregelter elektromagnetischer Antriebe für Luftschütze. Fortschr.-Ber. VDI Reihe 21 Nr. 293. Düsseldorf: VDI Verlag 2000

Quasi-Stationary Modeling and Simulation of Electrical Circuits using Complex Phasors

Anton Haumer Christian Kral Johannes V. Gragger Hansjörg Kapeller
 arsenal research
 Giefinggasse 2, 1210 Vienna, Austria
 anton.haumer@arsenal.ac.at

Abstract

This paper presents how complex phasors are used for quasi-stationary analysis of electrical circuits, i.e. with sinusoidal excitation neglecting dynamic transients. The theoretical background of complex phasors is elaborated and a Modelica implementation – the AC Library – is presented. Additional examples demonstrate the possibilities of the application of complex phasors.

Keywords: electrical circuit, sinusoidal excitation, quasi-stationary analysis, complex phasors

1 Introduction

In the simulation of physical systems described by a system of algebraic and ordinary differential equations we distinguish different types of simulation analysis:

- The *transient analysis* is the most general analysis, showing both the dynamic transients as well as steady-state solutions (if steady-state is reached).
- A *stationary analysis* (sometimes also called DC analysis) eliminates the derivatives with respect to time, determining steady-state solutions.
- The so-called *small signal AC analysis* linearizes a non-linear model in a certain point of operation (which is found by a stationary analysis), only applying excitations with small amplitudes.

Mainly in the field of electrical engineering – due to the nature of electrical power plants that provide nearly perfectly sinusoidal voltages with fixed frequency and amplitude – one more type of analysis is of great importance:

- *Quasi-stationary analysis* applies sinusoidal excitations with known frequency, amplitude and phase shift. In a circuit with isolated sub-circuits, each sub circuit may be operated at different fre-

quencies, however. Each frequency with respect to a sub-circuit is known due to the respective excitation. Fast dynamic transients are not considered. In a quasi-stationary analysis the unknown voltages and currents, with respect to their phase shift and amplitude, have to be determined. Regarding the consideration of exactly one frequency for each sub-circuit it has to be assumed that the only linear circuits are investigated.

This paper will demonstrate how complex phasors simplify the quasi-stationary analysis, and how complex phasors could be modeled using Modelica. Considering some limitations in the current Modelica version will lead to suggestions for improvement.

2 Complex Phasors

2.1 Representation of Sinusoidal Voltages and Currents

Any sinusoidal oscillation can be expressed by computing the real part of a complex time-dependent phasor according to Fig. 1:

$$a(t) = \hat{A} \cos(\omega t + \varphi) = \sqrt{2} \operatorname{Re}(\underline{A} e^{j\omega t}) \quad (1)$$

$$\underline{A} = A \cdot e^{j\varphi} \Rightarrow a(t) = \sqrt{2} \operatorname{Re}(\underline{A} \cdot e^{j\omega t}) \quad (2)$$

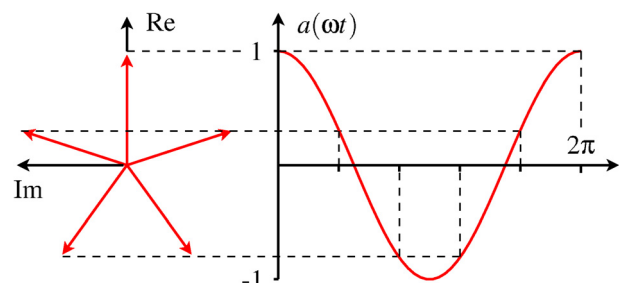


Fig. 1 The real part of a rotating phasor equals a sinusoidal oscillation; depicted phasor with $\varphi = 0$;

left: complex phasor $\sqrt{2} \underline{A} e^{j\omega t}$;

right: time domain signal $a(\omega t)$

The magnitude A of the complex phasor \underline{A} is the root mean square (RMS) value of the cosine wave. The phase shift φ is the phase shift of the cosine with respect to its maximum at $t = 0$. Time dependence is considered by the phasor $e^{j\omega t}$ and $\sqrt{2}$ is the ratio between the amplitude and the RMS value of the sinus waveform.

This background of complex phasors can also be applied to sinusoidal voltages and currents using complex voltage and current phasors:

$$v(t) = \hat{V} \cos(\omega t + \varphi_v) = \sqrt{2} \operatorname{Re}(\underline{V} \cdot e^{j\omega t}) \quad (3)$$

$$i(t) = \hat{I} \cos(\omega t + \varphi_i) = \sqrt{2} \operatorname{Re}(\underline{I} \cdot e^{j\omega t}) \quad (4)$$

Assuming sinusoidal excitation of an electric circuit, all voltages and currents are of sinusoidal waveform with the same angular frequency $\omega = 2\pi f$. Therefore the complex voltage phasor

$$\underline{V} = V \cdot e^{j\varphi_v} \quad (5)$$

and the complex current phasor

$$\underline{I} = I \cdot e^{j\varphi_i} \quad (6)$$

are sufficient to describe quasi-stationary voltages and currents.

The derivative of a complex phasor \underline{A} with respect to time leads to:

$$\frac{da(t)}{dt} = \sqrt{2} \operatorname{Re}(\underline{A} \cdot j\omega e^{j\omega t}) \quad (7)$$

The time derivative of a sinusoidal waveform is thus considered in the complex domain by multiplying the original phasor with $j\omega$. This relationship also implies the result of the integration with respect to the time domain. Since the constant of integration is zero for quasi-stationary analysis, the complex representation of a time domain integration is determined by the division of the original phasor by $j\omega$.

2.2 Modeling a Linear Resistor

A linear resistor can be described by the algebraic equation:

$$v = R \cdot i \quad (8)$$

Using complex phasors of voltage and current, we can replace the algebraic equation by a complex algebraic equation:

$$\underline{V} = R \cdot \underline{I} \quad (9)$$

2.3 Modeling a Linear Conductor

A linear conductor can be described by the algebraic equation:

$$i = G \cdot v \quad (10)$$

Using complex phasors of voltage and current, we can replace the algebraic equation by a complex algebraic equation:

$$\underline{I} = G \cdot \underline{V} \quad (11)$$

2.4 Modeling a Linear Inductor

A linear inductor can be described by the differential equation:

$$v = L \frac{di}{dt} \quad (12)$$

Exploiting the sinusoidal waveform of the current (4), we can replace the differential equation by a complex algebraic equation:

$$\underline{V} = j\omega L \cdot \underline{I} = \underline{X}_L \cdot \underline{I} \quad (13)$$

We find a complex version of the equation describing a resistor, using the complex reactance $\underline{X}_L = j\omega L$.

2.5 Modeling a Linear Capacitor

A linear capacitor can be described by the differential equation:

$$i = C \frac{dv}{dt} \quad (14)$$

Exploiting the sinusoidal waveform of the voltage (3), we can replace the differential equation by a complex algebraic equation:

$$\underline{I} = j\omega C \cdot \underline{V} = \underline{Y}_C \cdot \underline{V} \quad (15)$$

We find a complex version of the equation describing a conductor, using the complex admittance $\underline{Y}_C = j\omega C$.

2.6 Kirchhoff's Laws

For complex voltages and currents, respectively, Kirchhoff's Laws can be applied equivalently:

$$\sum I_i = 0 \quad (16)$$

The sum of all complex current phasors flowing to a node is zero.

$$\sum V_i = 0 \quad (17)$$

The sum of all complex voltage phasors in a closed loop is zero; this also implies that directly connected

nodes have the same complex potential. Both laws are inherently considered in Modelica connections.

2.7 Power

Multiplying a time dependent voltage (3) and the corresponding current (4), we obtain the instantaneous electrical power:

$$p(t) = v(t) \cdot i(t) \quad (18)$$

Substituting (3) and (4) in the electric power equation we obtain:

$$\begin{aligned} p(t) = & \\ & \sqrt{2}V \cos(\omega t + \varphi_V) \cdot \sqrt{2}I \cos(\omega t + \varphi_I) = \\ & V \cdot I \cdot [\cos(\varphi_V - \varphi_I) + \cos(2\omega t + \varphi_V + \varphi_I)] \end{aligned} \quad (19)$$

The instantaneous power oscillates with double the frequency of voltage and current, respectively. The average value of instantaneous power is dependent on the phase shift between voltage and current; this term is the active power:

$$P = V \cdot I \cdot \cos(\varphi_V - \varphi_I) = S \cdot \cos(\varphi_V - \varphi_I) \quad (20)$$

Apparent power S is defined as the product of the RMS values of the voltage and the current:

$$S = V \cdot I \quad (21)$$

Reactive power is defined as quadratic complement:

$$Q = \sqrt{S^2 - P^2} = S \cdot \sin(\varphi_V - \varphi_I) \quad (22)$$

Using complex phasors, we obtain:

$$\underline{S} = \underline{V} \cdot \underline{I} = P + jQ \quad (23)$$

In this equation, \underline{S} is the complex apparent power; the amplitude of this complex quantity is the apparent power (21).

3 Design of an AC Modelica Library

3.1 Implementation of Complex Arithmetics

Unfortunately complex numbers are not an intrinsic data type in the Modelica language. As a workaround, a record Complex containing both the real and the imaginary part of the complex number can be defined:

```
record Complex
  Real re "Real part";
  Real im "Imaginary part";
end Complex;
```

In some cases, the polar representation of a complex phasor, consisting of length and phase angle, is advantageous, however:

$$\underline{A} = A_{\text{Re}} + j \cdot A_{\text{Im}} = \hat{A} \cdot e^{j\varphi} \quad (24)$$

```
record Polar
  Real len "Length of the phasor";
  Modelica.SIunits.Angle phi "Phase angle";
end Polar;
```

Of course we have to provide functions for complex arithmetic + - * /, like

```
function '+' "Complex add"
  input Complex c1;
  input Complex c2;
  output Complex c3 "= c1 + c2";
algorithm
  c3 := Complex(c1.re + c2.re,
               c1.im + c2.im);
end '+';
```

as well as complex functions like

- abs length of the phasor
- arg phase angle
- conj conjugate complex
- sqrt square root
- exp natural exponentiation
- log natural logarithm
- sin sine
- cos cosine

which can be implemented according to a mathematical textbook.

It is not very elegant to use these functions:

```
v = Complex.'*(Complex(0, w*L), i);
```

Therefore an intrinsic implementation (or at least operator overloading) would allow reading, type and understanding code easier.

Additionally, we need conversion functions between rectangular and polar representation:

```
function fromPolar
  input Polar polar;
  output Complex result;
algorithm
  result.re := polar.len*cos(polar.phi);
  result.im := polar.len*sin(polar.phi);
end fromPolar;

function toPolar
  input Complex c;
  output Polar polar;
algorithm
  polar.len := Complex.'abs'(c);
  polar.phi := Complex.'arg'(c);
end toPolar;
```

The tricky part of the conversion from rectangular to polar representation is obtaining an angle that may be smoothly differentiated to obtain the angular velocity of the corresponding phasor. With the presented implementation the wrapping of the phase angle at 2π cannot be avoided. Instead, a continuous growth of the phase angle for non-zero frequency is desired.

A rather difficult exception of a smooth angle is the following example: Imagine a phasor with constant

angle, but length varying with time. The length shrinks within a certain time to zero, growing again in the opposite direction afterwards. This would lead to a discontinuity by π when the phasors crosses the origin.

Additionally, we have to define complex phasors with physical units, like:

```
record ComplexVoltage = Complex (
  redeclare Modelica.SIunits.Voltage re,
  redeclare Modelica.SIunits.Voltage im);
record ComplexCurrent = Complex (
  redeclare Modelica.SIunits.Current re,
  redeclare Modelica.SIunits.Current im);
```

to take advantage of a tool's type checking capabilities. Furthermore we have to define the polar representations, too:

```
record PolarVoltage = Polar (
  redeclare Modelica.SIunits.Voltage len);
record PolarCurrent = Polar (
  redeclare Modelica.SIunits.Current len);
```

3.2 Propagation of the Common Frequency

Since different sub-circuits of an electrical circuit could have different frequencies – e.g. stator and rotor of an asynchronous induction motor– it would be advantageous to provide the local frequency of a component via the connector. Introducing an additional variable (reference angle, frequency or angular velocity) in the connector leads to over-determined connection equations. Fortunately, Modelica [4] provides methods to deal with this problem. This connector variable has to be defined as a type or record with an additional function definition:

```
record Reference
  Modelica.SIunits.Angle phi;
  function equalityConstraint
    input Reference ref1;
    input Reference ref2;
    output Real residue[0];
  algorithm
    residue := ...;
  end equalityConstraint;
end Reference;
```

Additionally, the following functions are used to allow a tool to break algebraic loops:

- `Connect`
defines a breakable branch
- `Connections.branch`
defines a non-breakable branch
- `Connections.root`
defines a root node in a virtual connection graph
- `Connections.potentialRoot`
defines a potential root node in a virtual connection graph

3.3 Single Phase Components

The connector definition

```
connector Pin
  Types.ComplexVoltage v;
  flow Types.ComplexCurrent i;
  Types.Reference ref;
end Pin;
```

not only contains complex potential and complex current, but also the record providing the local frequency respectively phase angle of the reference frame as explained in 3.2.

Additionally, basic components as ground, resistor, conductor, capacitor and inductor are defined. Furthermore, we need sensors and voltage sources as well as current sources. Fig. 2 gives an overview of the implemented components.

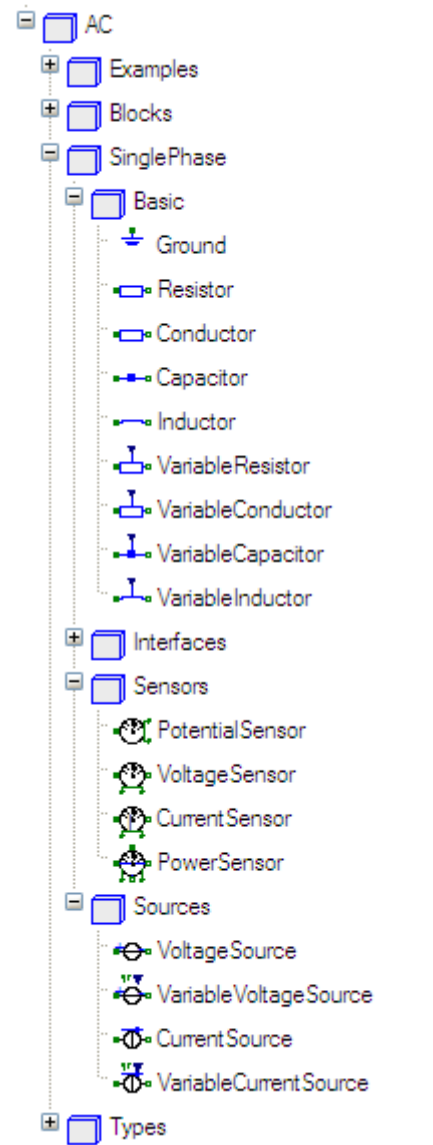


Fig. 2 Structure of the AC library

As an example, the implementation of the inductor as well as the partial models that inductor extends from are shown:

```
partial model TwoNode
  Types.ComplexVoltage v =
    Complex.'-'(p.v, n.v);
  Types.ComplexCurrent i = p.i;
  Modelica.SIunits.AngularVelocity w =
    der(p.ref.phi);
  AC.SinglePhase.Interfaces.PositivePin p;
  AC.SinglePhase.Interfaces.NegativePin n;
equation
  Connections.branch(p.ref, n.ref);
  p.ref.phi = n.ref.phi;
end TwoNode;
```

`TwoNode` defines the complex voltage drop along the component as well as the angular velocity by differentiating the reference phase angle.

```
partial model OnePort
  extends TwoNode;
equation
  Complex.'+'(p.i, n.i) = Complex.'0'();
end OnePort;
```

`OnePort` additionally defines that the sum of currents flowing into the component is zero.

```
model Inductor
  extends Interfaces.OnePort;
  parameter
    Modelica.SIunits.Inductance L=1;
equation
  v = Complex.'*(Complex(0, w*L), i);
end Inductor;
```

Using these partial models `Inductor` is a simple implementation of (13).

3.4 Auxiliary Blocks

Additionally to the basic components, blocks with complex inputs / outputs are needed. Therefore a complex signal is defined, as well as a polar signal:

```
connector ComplexSignal = AC.Types.Complex;
connector PolarSignal = AC.Types.Polar;
```

These connectors are used to define `ComplexInput`, `ComplexOutput`, `PolarInput` and `PolarOutput`. Instances of these output signal connectors are needed for sensors, as well as input signal connectors for variable sources.

Additionally some useful blocks are defined:

- `ToComplex` generates a complex phasor from real inputs, either real and imaginary part or amplitude and phase angle.
- `FromComplex` generates real outputs – real and imaginary part as well as amplitude and phase angle – either from a complex input or a polar input.
- `ToPolar` takes a complex input and generates a polar representation of the phasor as the output.

- `FromPolar` takes a polar representation of a complex phasor on the input and generates a complex phasor as the output.
- `FromPolar` calculates the complex sum of an array of complex input phasors.

4 Simulation Examples

For calculating quasi-stationary characteristic curves of an electrical circuit varying a parameter the usage of the AC library is advantageous. This will be demonstrated on four examples:

- Current of a series resonance circuit, varying the supply frequency
- Voltage of a parallel resonance circuit, varying the supply frequency
- Torque and current of an asynchronous induction machine, varying slip
- Terminal voltage of a synchronous induction machine, varying load impedance (resistive and inductive).

4.1 Series Resonance Circuit

As a first example, we model a series resonance circuit (Fig. 3).

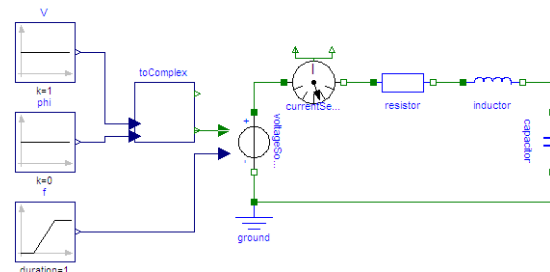


Fig. 3 Model of a series resonance circuit

We apply sinusoidal voltage with constant amplitude and phase to a series connection of a resistor, an inductor and a capacitor. Frequency varies according to a ramp.

Analytically the resonant frequency of this simple experiment can be determined:

$$\omega_{\text{res}} = \frac{1}{\sqrt{LC}} \quad (25)$$

With $L = \frac{1}{2\pi}$ H and $C = \frac{1}{2\pi}$ F we derive a resonance frequency at $f = 1$ Hz. From the amplitude (Fig. 4) as well as the phase shift (Fig. 5) of the current, the resonance frequency is evident.

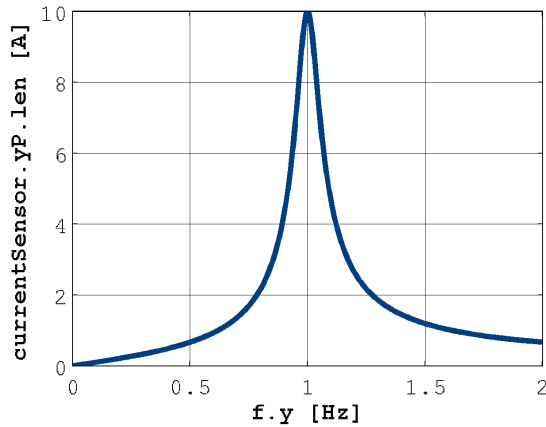


Fig. 4 Amplitude of current versus excitation frequency

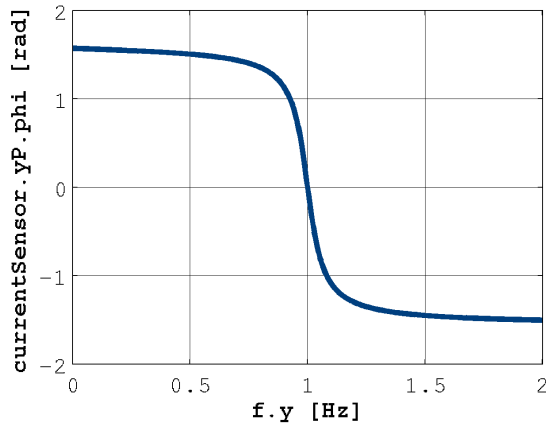


Fig. 5 Phase shift of current versus excitation frequency

4.2 Parallel Resonance Circuit

Furthermore, we investigate a parallel resonance circuit (Fig. 6).

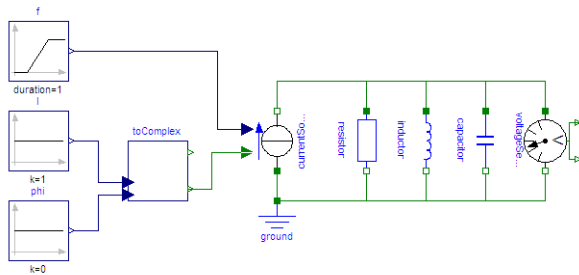


Fig. 6 Model of a parallel resonance circuit

We inject a sinusoidal current with constant amplitude and phase to a parallel connection of a resistor, an inductor and a capacitor. Frequency varies according to a ramp. The resonant frequency of the parallel resonant circuit is:

$$\omega_{\text{res}} = \frac{1}{\sqrt{LC}} \quad (26)$$

With $L = \frac{1}{2\pi}$ H and $C = \frac{1}{2\pi}$ F we derive a resonance frequency at $f = 1$ Hz. From the amplitude (Fig. 7) as well as the phase shift (Fig. 8) of the voltage, the resonance frequency is evident.

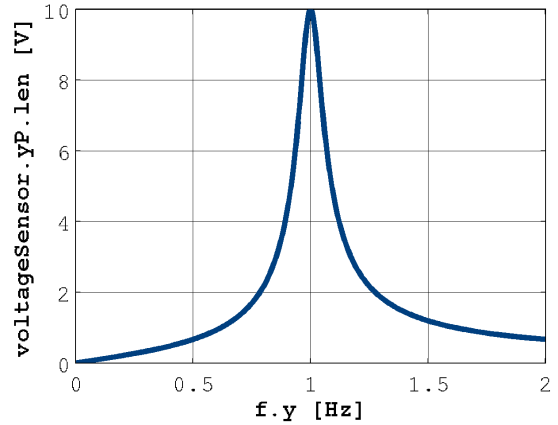


Fig. 7 Amplitude of voltage versus excitation frequency

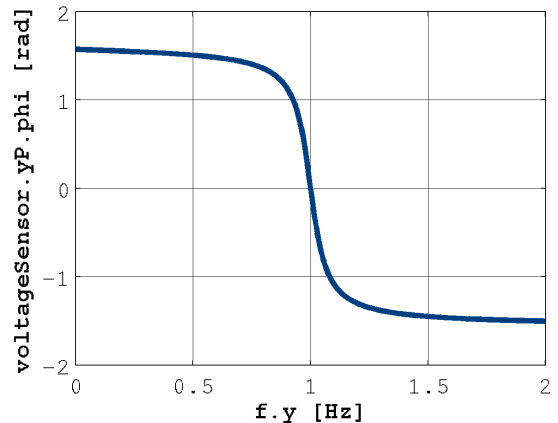


Fig. 8 Phase shift of voltage versus excitation frequency

4.3 Asynchronous Induction Machine

Quasi-stationary operation of a three-phase asynchronous induction machine with squirrel cage (AIMC) may be described by an equivalent circuit as depicted in Fig. 9. This equivalent circuit represents one phase of a symmetrical three phase asynchronous induction machine, however.

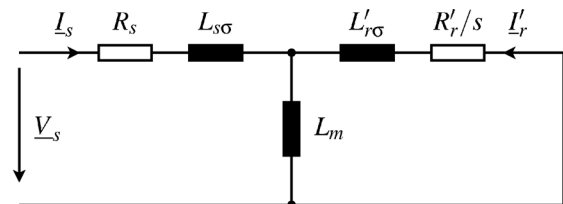


Fig. 9 Single phase equivalent circuit of an AIMC

In this equivalent circuit R_s is the stator resistance, $L_{s\sigma}$ is the stator leakage inductance, and L_m is the main field inductance. In the rotor circuit $L_{r\sigma}'$ is the rotor leakage inductance and R_r' is the rotor resistance. Both these rotor components refer to an equivalent stator winding and are thus indicated by '. An implementation of this equivalent circuit in Mod- elica is shown in Fig. 10.

For an induction machine slip

$$s = 1 - \frac{\omega}{\omega_s} \quad (27)$$

is the relative deviation of the mechanical angular velocity ω from the synchronous angular velocity:

$$\omega_s = \frac{2\pi f}{p} \quad (28)$$

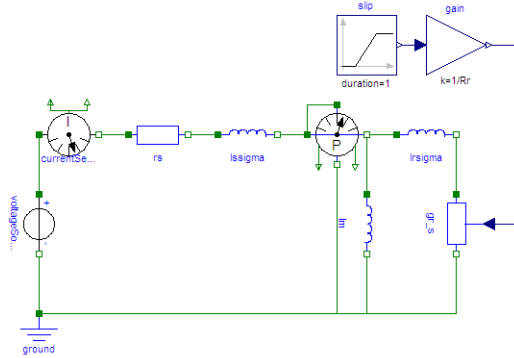


Fig. 10 Model of an AIMC

In the presented example slip is modeled as a ramp from slip 100% (i.e. stand-still) to slip 0% (i.e. no-load). Dividing the rotor resistance by the slip is equivalent to multiplying the rotor conductance by slip. The slip dependent rotor conductance is thus modeled by a variable conductance $g_{r,s}$. Using the conductance avoids division of zero slip at no-load:

$$R'_{r,actual} = \frac{R'_r}{s} \quad (29)$$

The motor parameters used for this example are the same as those of the dynamic model `Electrical.Machines.BasicMachines.AsynchronousInductionMachines.AIM_SquirrelCage`.

This leads to the quasi-stationary motor characteristics depicted in Fig. 11 and Fig. 12. The horizontal axis of these plots shows the relative (per unit) speed which is equal to $(1-slip)$.

Fig. 12 shows only 1/3 of the total air gap power of the machine since only one phase is modeled. The total airgap torque can thus be determined by:

$$T = \frac{3 \cdot P_{airgap}}{\omega_s} \quad (30)$$

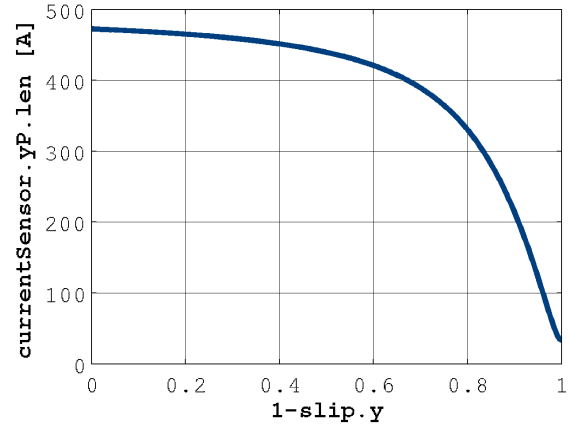


Fig. 11 Stator current versus $(1-slip)$

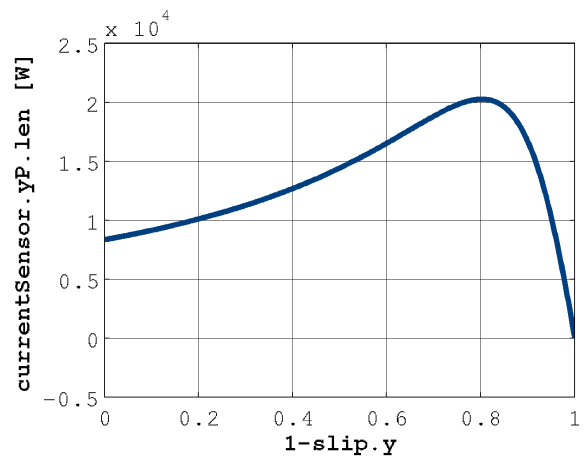


Fig. 12 Airgap power versus $(1-slip)$

4.4 Synchronous Induction Machine

A synchronous induction machine feeding an isolated system is presented in this example. Two cases are investigated: resistive load (Fig. 13) and inductive load (Fig. 14).

In both cases, constant excitation is assumed. Synchronous induced voltage is modeled by a voltage source with constant complex voltage phasor. l_d represents the synchronous reactance and r_s the resistance of one phase. Variable load is prescribed by a ramp with logarithmic scale.

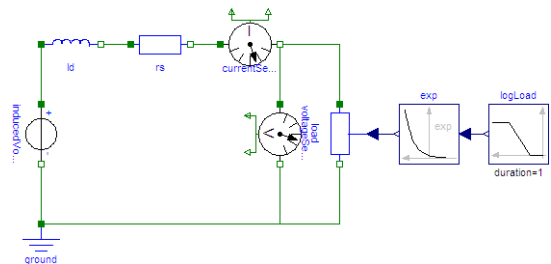


Fig. 13 Synchronous induction machine with R-load

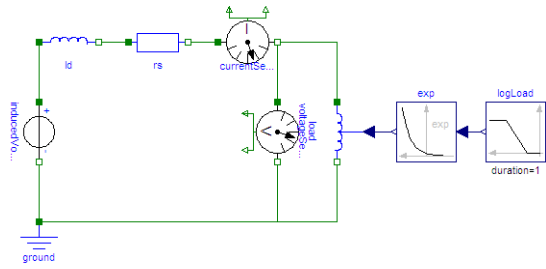


Fig. 14 Synchronous induction machine with L-load

Fig. 15 shows the characteristic voltage versus current from nearly no-load (high resistance and inductance, respectively) to nearly short circuit (low resistance and inductance, respectively).

The machine parameters used for this example are the same as those of the dynamic model `Electrical.Machines.BasicMachines.SynchronousInductionMachines.SM_ElectricalExcited`.

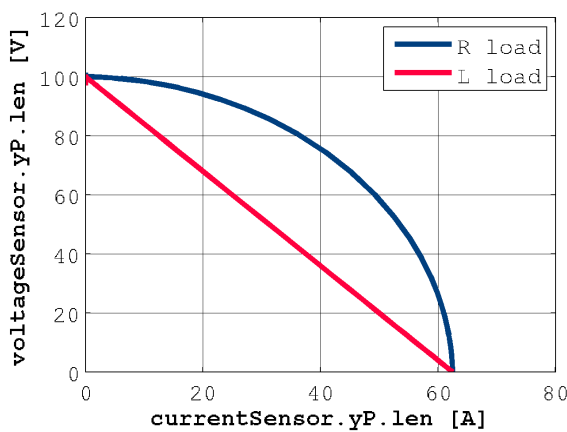


Fig. 15 Voltage versus current for resistive load and inductive load

5 Conclusions and Outlook

The design of a Modelica library for quasi-stationary analysis of electrical single-phase circuits has been presented. The application of complex algebraic equations instead of dynamic differential equations leads to high performance simulations.

With respect to the current Modelica version 3.0, the implementation of complex numbers is possible but not really satisfying. The authors would suggest the introduction of complex numbers as an intrinsic data type. This data type and complex arithmetics would improve the Modelica language, however.

Based on the presented draft of an AC library, the next steps will be extending the components for multi-phase circuits as well as modeling of asyn-

chronous and synchronous induction machines for quasi-stationary analysis. These machine models are planned to be based on space phasors as described in [6]; the transformation of space phasors with respect to different reference frames has to be implemented.

For applications focused on the energy consumption of an electric drive over a longer period of time, the fast electrical transients can be neglected. Using complex quasi-stationary models would lead to faster simulations, however.

References

- [1] M. L. Boas, *Mathematical Methods in the Physical Sciences*. J. Wiley & Sons 1966
- [2] T. D. Burton, *Introduction to Dynamic Systems Analysis*. McGraw Hill 1994
- [3] R. C. Dorf, *The Electrical Engineering Handbook*. VDE 1993
- [4] Modelica Specification, version 3.0. <http://www.modelica.org/documents/ModelicaSpec30.pdf>
- [5] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Piscataway, NJ: IEEE Press, 2004.
- [6] C. Kral, A. Haumer, *Modelica libraries for dc machines, three phase and polyphase machines*. 4th International Modelica Conference 2005, Hamburg, Germany
- [7] O. Enge, C. Clauß, P. Schneider, P. Schwarz, M. Vetter, S. Schwunk, *Quasi-stationary AC Analysis Using Phasor Description With Modelica*. 5th International Modelica Conference 2006, Vienna, Austria

Session 3a

Language, Tools and Algorithms



HyAuLib: modelling Hybrid Automata in Modelica

Tiziano Pulecchi Francesco Casella
 Politecnico di Milano, Dipartimento di Elettronica e Informazione
 Piazza Leonardo da Vinci 32, 20133 Milano, Italy

Abstract

A library of components for modelling hybrid automata in a natural fashion has been implemented in Modelica. This library exploits and extends the free Modelica library StateGraph to the modelling and simulation of deterministic hybrid systems described by the hybrid automaton formalism. In this contribution the library's main features are described and its flexibility highlighted by developing models for two classic hybrid systems literature examples.

Keywords: hybrid automata; simulation

1 Introduction

Hybrid systems (see [7]) are dynamical systems involving the interaction of both continuous state and discrete state dynamics. Recall that a state variable is called discrete if it can take a finite (or countable) number of values and continuous if it takes values in the Euclidean space R^n for some $n \geq 1$. By their nature, discrete states can change value only through a discrete *jump*; on the other hand, continuous states can change values either through a *jump*, or by *flowing* in time according to a given differential equations set.

Physical systems are by their own nature inherently continuous. Nevertheless, because of the couplings with very high frequency dynamics, or in presence of mechanisms too complicated to be dealt with in simulation by a sound physical description, many physical systems can be conveniently represented under the hybrid systems paradigm. This provides a convenient framework for modelling systems in a wide range of engineering applications, including for instance electrical circuitry, where continuous dynamic is affected by switches opening and closing; chemical processes control, where the continuous evolution of chemical reactions is controlled by valves and pumps; or digital control, where digital computers interact with a continuous time physical system. Of course, highly non-

linear systems such as for instance diodes, switches, valves, mechanical backlashes and dead strokes, can be conveniently described via abstracted hybrid models.

The analysis and design of hybrid systems is in general more demanding than that of purely discrete or purely continuous systems, because of the necessity to accurately deal with the interplay between the discrete and continuous dynamics. The same consideration holds true for their simulation, that presents specific challenges requiring special care. Specifically, it is of paramount importance to be able to determine with great accuracy the time instant when discrete jumps take place, and consistently deal with simultaneous events occurrences, which represents one of the main sources of modelling inconsistencies.

Nowadays, general purpose simulation packages such as Matlab and Simulink can deal adequately with most complications. Specialized packages have also been developed that allow accurate simulation of hybrid systems (see *e.g.* [2], [3], [1], [5]). The interested reader is addressed to [4] for a thorough overview on the subject.

In this paper a library of components for modelling (autonomous) hybrid automata (HyAuLib), implemented in Modelica (see [9, 6]), has been designed. Modelica is already capable of efficiently handle hybrid systems modelling and simulation via suitable scripts (see [8]). Nevertheless, sometimes this operation can turn out to be very cumbersome and error prone for the unexperienced user. This library, extending the free Modelica library StateGraph (see [10]), overcomes these difficulties by providing an easy way to the consistent modelling and simulation of hybrid systems described by the hybrid automaton formalism.

The paper is organized as follows: in Sections 2 and 3 the hybrid automaton formalism and the implemented Modelica HyAuLib will be respectively described. In Section 4, the library capabilities shall be illustrated on two classic hybrid systems textbook examples. Fi-

nally, in Section 5 concluding remarks and future developments will be presented.

2 Hybrid Automata

A hybrid automaton is a dynamical system describing the evolution in time of a set of discrete and continuous variables. In this paper we will focus on autonomous hybrid automata, *i.e.* hybrid automata which have no inputs nor outputs. More specifically, the transitions between two modes of our automata shall occur in accordance with a user-specified determinism. This topic will be thoroughly discussed in Section 3. The hybrid automaton will answer to the following definition (see *e.g.* [7]):

Definition 2.1 (Hybrid Automaton) A hybrid automaton H is a collection $H = (Q, X, f, Init, D, E, G, R)$ where

- $Q = \{q_1, q_2, \dots\}$ is the set of all admissible discrete states, or *modes* of H ;
- $X \subseteq R^n$ is the set of continuous states;
- $f(\cdot, \cdot) : Q \times X \rightarrow R^n$ is a *vector field*, defining the evolution in time of the continuous part of the state of H ;
- $Init \subseteq Q \times X$ is the set of all admissible initial states for H ;
- $Inv(\cdot) : Q \rightarrow P(X)$ is the *invariant set* or domain;
- $E \subseteq Q \times Q$ is a set of *edges*, defining all transitions from one mode of H to the next;
- $G : E \rightarrow P(X)$ is the set of *guard* conditions;
- $R(\cdot, \cdot) : E \times X \rightarrow P(X)$ is a *reset map*,

where $P(X)$ denotes the power set (the set of all possible subsets of X), and the pair $(q, x) \in Q \times X$ is the state of H , made up by its discrete and continuous contributors.

Hybrid automata define possible evolutions for their state. Roughly speaking, starting from an initial value $(q_0, x_0) \in Init$, the continuous state x flows according to the differential equation

$$\begin{cases} \dot{x} = f(q_0, x) \\ x(0) = x_0 \end{cases}$$

defined by the hybrid automaton vector field, while the discrete state q remains constant, *i.e.*,

$$q(t) = q_0$$

as long as $x \in Inv(q_0)$. If at some point x reaches the guard $G(q_0, q_1) \subseteq R^n$ of some edge $(q_0, q_1) \in E$, then the discrete part of the state q may change to q_1 . If this happens, x is reset according to the reset map $R(q_0, q_1, x)$. After a discrete transition has taken place, continuous evolution resumes, until a new transition is triggered, and so on.

To define the time horizon over which the states of the hybrid system evolve, we need to introduce the concept of Hybrid Time Set by the following definition (see *e.g.* [7]):

Definition 2.2 (Hybrid Time Set) A hybrid time set is a sequence of contiguous intervals $\tau = \{I_0, I_1, \dots, I_n\}$ finite or infinite such that

- $I_i = [\tau_i, \tau'_i]$ for all $i < n$;
- if $N < \infty$ then either $I_N = [\tau_N, \tau'_N]$ or $I_N = [\tau_N, \tau'_N)$;
- $\tau_i \leq \tau'_i = \tau_{i+1}$ for all i ,

where τ'_i represents the time instant immediately preceding a discrete transition occurrence, whereas τ_{i+1} corresponds to the time instant just following the discrete transition. The adoption of this representation of time in hybrid automata allows the handling of situations where multiple transitions occur simultaneously. When the range of a generic interval I_j shrinks to a single value τ_j , it means that the associated mode Q_k has been entered and exited in the very same instant. If multiple transitions are enabled and do occur simultaneously, the automaton evolves to its new mode Q_r passing through several intermediate modes, whose associated residing times are zero.

The triple (τ, q, x) consisting of a hybrid time set and two sequences of functions $q = \{q_i\}$ and $x = \{x_i\}$ is named a *hybrid trajectory*, whereas an *execution* of H is a hybrid trajectory (τ, q, x) admissible by the hybrid automaton H .

Note in passing that the exploitation of the invariant set definition, allowable within the hybrid automaton formalism, could be used to efficiently simulate a broad variety of *unconventional* engineering applications, such as, for instance, the suitability of the designed safety procedures for continuous dynamical systems. This could be achieved simply by modelling the safety critical conditions for the system via undesirable regions for the continuous state (by defining the automaton domain accordingly). As a consequence, if the recovery procedures fail, the automaton will violate the invariant set and the simulation will be terminated with a warning message specifying the safety

critical condition violated, leading to a procedure re-design. Many other useful controls of this kind, such as for example *Zeno behavior* detection, can be easily incorporated into the Modelica HyAuLib.

Another interesting field of application for the HyAuLib could be in the framework of simulation of systems undergoing failures. Let's focus on a simple example, where a relief valve is used to control the pipeline pressure in an hydraulic plant. During nominal operational regime, the valve remains closed while pressure at the valve inlet is lower than the valve preset pressure. When the preset pressure is reached, the valve is opened and the pressure at the inlet reduced. Now, if the valve experiences a failure and got stuck in the open position, and for some operational reason the pressure in the pipeline crosses the valve preset pressure and keeps rising, either the valve's backup (if any) will be activated, or the system will suffer damage and loose functionality. If no backup is activated, or it results ineffective, the pressure will utterly increase to the point of exceeding a new threshold value (defining when the system is no more operative), specified via the invariant set. The simulation will be either terminated if the experienced failure is classified as safety significant or safety critical (the system architecture needs a redesign), or kept running with the system in failure. Note that our simple example requires that the transition between the operational modes of the pressure relief valve (open and close) is triggered either by an opening (*resp.* closing) command or as a consequence of a failure experienced by the equipment and associated to a probability of occurrence. Relevant data both in terms of failure modes and failure rates can be obtained from the equipment's Failure Mode Effect and Criticality Analysis document, and the necessary hybrid models easily implemented exploiting the HyAuLib models, described in the following Section 3.

3 The Modelica HyAuLib

The Modelica HyAuLib addresses the problem of supporting the designers working with hybrid systems, by providing them with an efficient and intuitive modelling and simulation tool for hybrid automata. The library has been derived by extending the free Modelica StateGraph library by Otter and Dressler (see [10]), which is based on the JGraphChart method and provides components to model finite state machines. The HyAuLib allows for the modelling of complex hybrid systems that can be represented throughout the

hybrid automata paradigm in a natural fashion. Although such models could of course be obtained by writing explicitly the relevant Modelica code, this task is likely to turn out to be burdensome and error prone even for very simple models. The HyAuLib, by automatically managing all state transitions and the discrete/continuous domains interaction, seeks to minimize all possible sources of wrong modelling behavior. An expanded view of the HyAuLib tree is given in Figure 1.

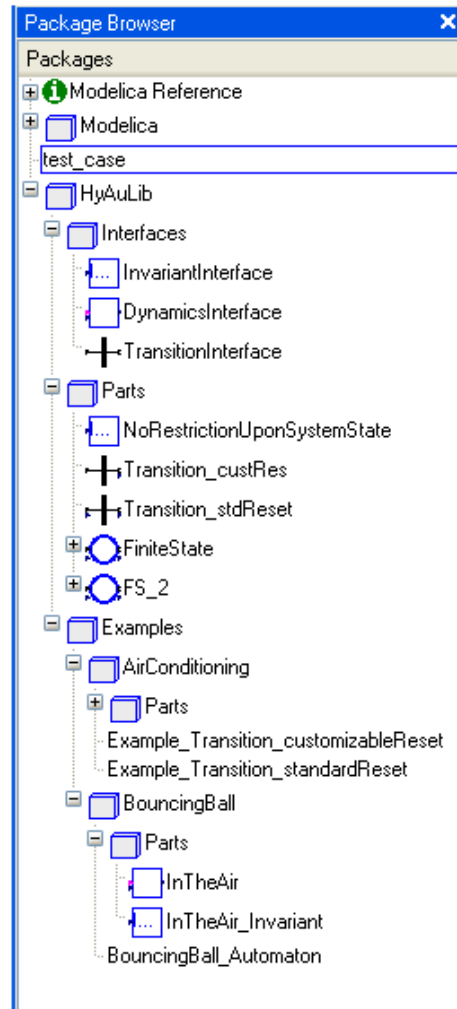


Figure 1: The Modelica HyAuLib library's tree.

The Modelica HyAuLib encompasses two basic components, the *FiniteState* and *Transition* models, which are briefly summarized in the following:

- Component *FiniteState*: defines each finite state (or mode) for the generic automaton. Embedded in this model are the definitions of the vector field and of the invariant set associated to the present state of the hybrid automaton. Component *FiniteState* extends StateGraph's *Step* com-

ponent, which is used to define which state is presently active. (See Figure 2, where the Modelica code used to generate the *FiniteState* component is shown.)

The *FiniteState* options selection mask is shown in Figure 3. For every *FiniteState* component instantiated in the hybrid automaton model, the number of input and output connections needs to be specified, jointly with the selection between StateGraph's InitialStep and Step components (for further information on StateGraph, see [10]). Finally, notice that a given time continuous dynamics and invariant set must necessarily be specified. These models can be elementarily defined by extending suitable interfaces provided within the HyAuLib library. Several of such examples are provided in the library's *Example* folder.

- Component *Transition*: defines the generic transition between modes. The model, extending StateGraph's *Transition* component, comprises the definition of the guard and the reset conditions. Notice that all transitions are triggered according to a deterministic mechanism, *i.e.*, at the moment no provision is given within the library in order to assign a probability to the transition. Two possible reset conditions are available: the standard option guarantees the continuity of the continuous state variable throughout the transition, whereas the second option allows for the definition of a specified reset. The Modelica code for the *Transition* model is provided in Figure 4. Also, Figure 5 shows the options selection mask for the HyAuLib *Transition* component.

Both models take full advantage of the Modelica *redeclare* construct feature, which makes it possible to create general classes which are defined only when the model is instantiated. It is then possible, once suitable models for the hybrid automaton's dynamics, invariant sets, guards and reset conditions have been defined, to simply drag and drop in the automaton model the base *FiniteState* and *Transition* models, select the relevant features from the graphical user interface, and connect them to reproduce the automaton scheme.

The HyAuLib supports multiple edges connection between different modes (the number of input and output connections being a parameter of the *mode* component). The transition mechanism adopted is deterministic. The transition is triggered as soon as the guard condition is satisfied, or with a delay that can be specified as a function of the hybrid automaton state and

current time. Future development will comprise the implementation of a probabilistic approach in the definition of the transition occurrence.

Notice that no care needs to be taken in the HyAuLib with respect to the definition of an hybrid time set for the automaton execution. Modelica is indeed capable of dealing with this issue, requiring no further modelling endeavor.

4 Case studies

In the following, two classic applications, which have been extensively discussed in the hybrid systems control literature, are presented to illustrate the HyAuLib capability when modelling hybrid automata.

4.1 Bouncing ball

A bouncing ball is a very effective example of an highly nonlinear dynamical system, which can be conveniently represented as a simple hybrid automaton with a single discrete state, describing the ball being above the ground. Here, all the system nonlinearities are easily modelled by introducing a hybrid component in the model.

The system state is a two dimensional vector comprising the ball's center of gravity height from ground and its derivative, the vertical velocity. The state continuous time evolution is then described by

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -9.81 \end{cases}$$

where x_1, x_2 are the vertical position and velocity respectively, and the associated invariant is given by the condition $x_1 \geq 0$ (ball in the air).

The only transition possible occurs from the state to itself when the ball hits the ground: the associated guard condition is then

$$x_1 = 0 \text{ and } x_2 \leq 0.$$

A nonconservative description of the phenomenon may be easily accounted for by acting upon the reset condition. Proceeding like that, we could easily force an energy loss due to the deformation of the system simply by setting

$$x_2 := -cx_2$$

with c non negative and less than unity.

```

model FiniteState
  "Interface for the generic finite state of the Hybrid Automaton"

  parameter Integer nIn(min=0)=1 "Number of input connections";
  parameter Integer nOut(min=0)=1 "Number of output connections";

  replaceable model StepType = Modelica.StateGraph.StepWithSignal
    extends StepType
    "choice between StaeGraph's InitialStep and Step (default) blocks" #;

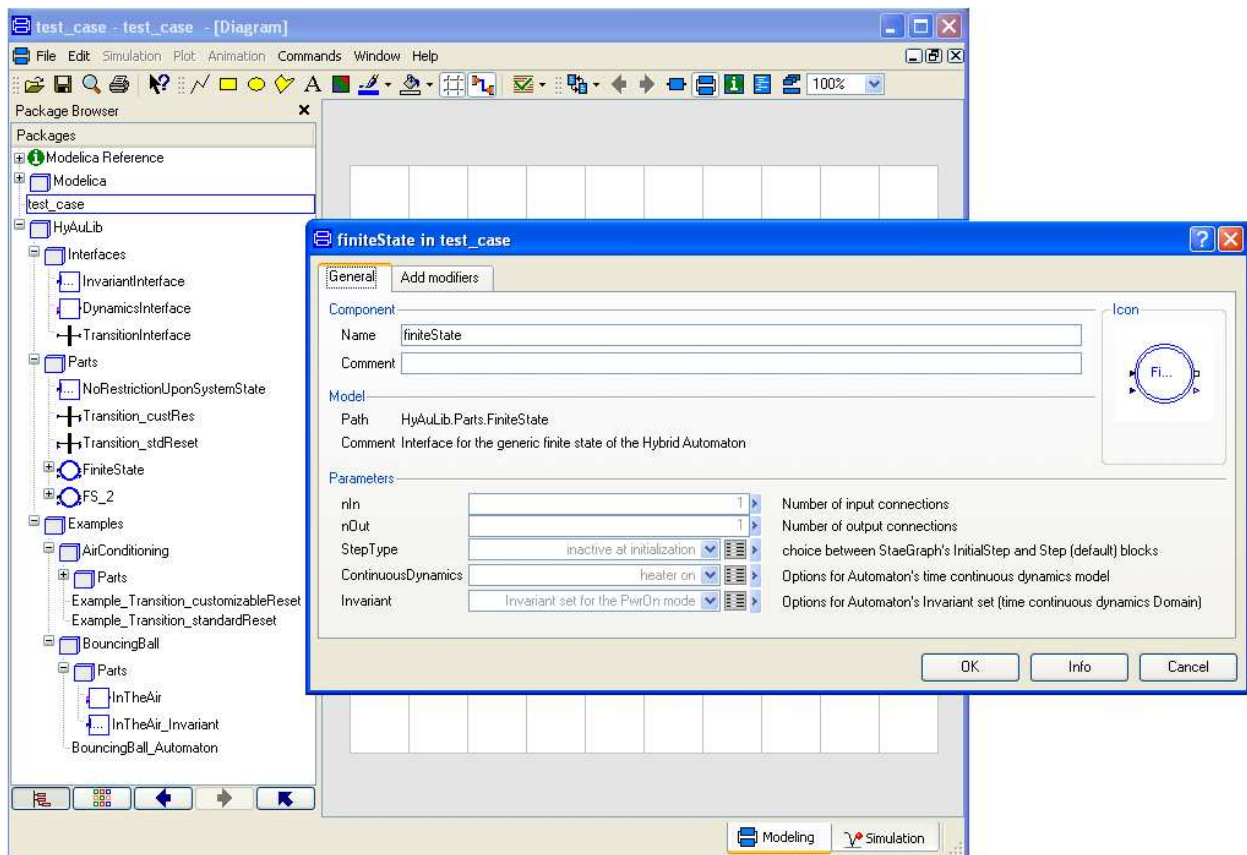
  replaceable model ContinuousDynamics =
    HyAuLib.Examples.AirConditioning.Parts.Dynamics.PowerON extends
    HyAuLib.Interfaces.DynamicsInterface
    "Options for Automaton's time continuous dynamics model" #;

  replaceable model Invariant =
    HyAuLib.Examples.AirConditioning.Parts.Invariants.PwrOn_Invariant
    extends HyAuLib.Interfaces.InvariantInterface
    "Options for Automaton's Invariant set (time continuous dynamics Domain)" #;

equation

end FiniteState;

```

Figure 2: Modelica code of the *FiniteState* component.Figure 3: User's selection mask for the HyAuLib library's *FiniteState* component.

```

model Transition_custRes
  "Model for transition from finiteState_In to finiteState_Out with customizable reset"

  extends Interfaces.TransitionInterface;

  // reset settings
  parameter Integer nOut(min=1)=1 "cardinality of finiteState_Out state";
  parameter Real[nOut] resetValue=zeros(nOut)
    "chosen reset value for finiteState_Out state";

equation
  reset = resetValue;

end Transition_custRes;

```

Figure 4: Modelica code of the *Transition* component.

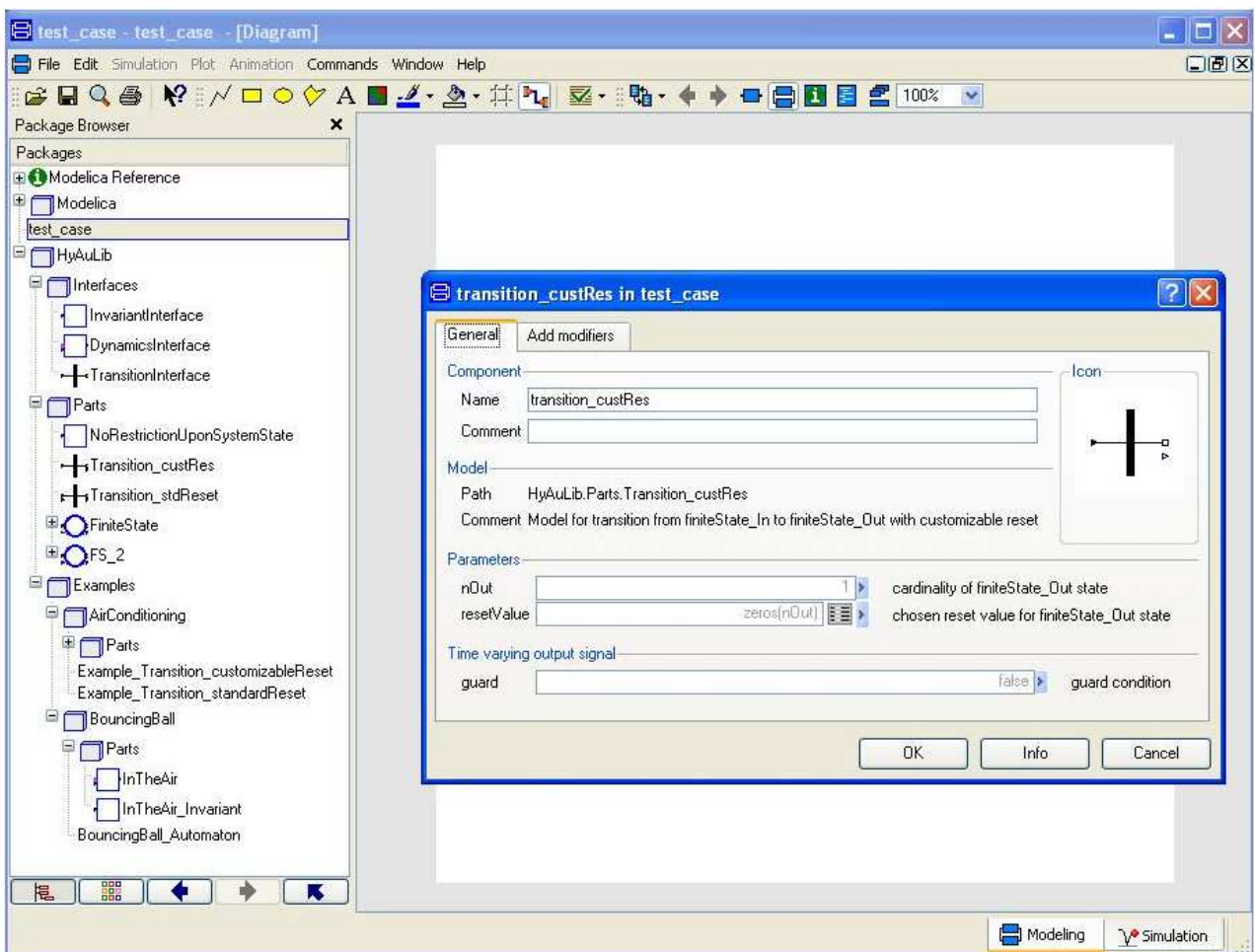


Figure 5: User's selection mask for the HyAuLib library's *Transition* component.

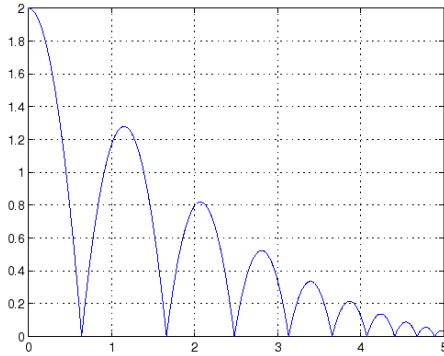


Figure 6: Vertical position of the bouncing ball vs. time. HyAuLib model.

Figure 6 shows the evolution in time of the ball position, given an initial height of 2 meters, a null initial velocity and a damping coefficient $c = 0.8$.

Both a Simulink/Stateflow and a Modelica flat model for the bouncing ball were realized, to serve as a reference for a discussion about the HyAuLib modelling performance. Both models (provided that the zero-crossing block is used in Simulink) provide good accuracy as long as the ball's energy is sufficiently large. Anyway, the *Zeno behavior* typical of this example, cause a severe impair of performance when the ball's vertical position x_1 gets very small. Due to numerical errors, x_1 will eventually become negative and, since the equations used to describe the model are still satisfied, the ball position will keep decreasing. This behavior, depicted in Figure 7, corresponds to the ball passing through the floor and keep falling, and is of course not admissible. This problem is naturally avoided if the HyAuLib's components are used, since the specification of the hybrid automaton invariant set clearly marks negative values for x_1 as unfeasible.

4.2 Air conditioning system

Let's now consider the problem of designing a room air conditioning system. We want to keep the room temperature within a specified range by acting upon a heating device. Assume that the desired temperature is 19 degrees centigrade, and the thermostat policy is to turn the heater on whenever the room temperature drops below 17°C, and turn it off when it passes 21°C. For simplicity's sake let the room temperature evolution in time be subjected to the simplified law

$$\dot{T} = -0.05T + 1.5\delta,$$

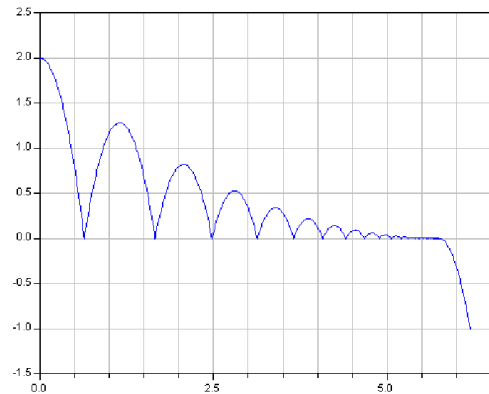


Figure 7: Vertical position of the bouncing ball vs. time. Modelica flat code.

where $\delta = 0$ if the heater is turned off and $\delta = 1$ if the heater is on. The hybrid automaton will then comprise two modes and two transitions, which can be defined as follows:

1. Hybrid Automata modes:

a) heating on ($q = ON$)

The continuous state evolves according to

$$\dot{T} = -0.05T + 1.5 \quad (1)$$

whereas the invariant set is $T \leq 21$.

b) heating off ($q = OFF$)

The continuous state evolution is given by

$$\dot{T} = -0.05T \quad (2)$$

whereas the invariant set is $T \geq 17$.

2. Transitions:

a) from ON to OFF

The guard condition is $T \geq 21$,

whereas the reset condition is $T := 21$.

b) from OFF to ON

The guard condition is $T \leq 17$,

whereas the reset condition is $T := 17$.

Notice how the evolution of the continuous and discrete states of the automaton are tightly coupled. Whenever $q = ON$, the temperature rises according to (1), whereas it decays according to (2) when $q = OFF$. Likewise, the evolution in time of the discrete state is constrained by the continuous state value: it cannot jump from ON to OFF or viceversa unless the guard condition is triggered.

Figure 8 shows the evolution in time of the room air temperature and the periodic switching of the heater from power on to power off and viceversa. The air conditioning system was initialized in power off, with a room temperature of 14°C. Whenever the temperature upper bound (21°C) is reached, the heater is powered off, and the room starts cooling until the lower bound for the admissible temperature (17°C) is hit. Then, the heater is powered on and a new cycle begins.

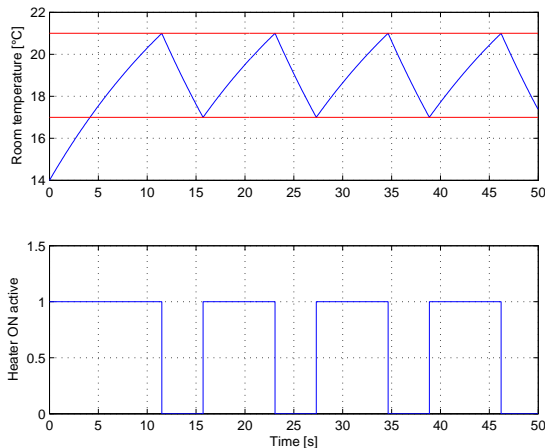


Figure 8: Room temperature and Heater ON status vs. time.

5 Concluding Remarks

In this paper the HyAuLib, a Modelica library for modelling and simulation of autonomous hybrid automata, extending the free Modelica StateGraph library for Finite State Machines, has been presented, and its main features illustrated throughout the simulation of two classic hybrid system textbook case studies. The HyAuLib allows, even to the most unexperienced user, to derive in a natural way models for simulating complex hybrid systems. Future developments of the HyAuLib will comprise the inclusion of a probabilistic approach with respect to the transition occurrence and the exploration of the library's capabilities and efficiency in modelling complex applications in the system safety design area.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, I. Lee. Modular Specification of Hybrid Systems in Charon. Technical Memorandum,

<http://www.cis.upenn.edu>, University of Pennsylvania, Philadelphia, PA.

- [2] M. Anderson. Object-Oriented Modeling and Simulation of Hybrid Systems. Ph.D thesis, Lund Institute of Technology, Lund, Sweden, December 1994.
- [3] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng. HyVisual: A Hybrid System Visual Modeler, Technical Memorandum, UCB/ERL M05/24, <http://ptolemy.eecs.berkeley.edu/publications/papers/05>, University of California, Berkeley, CA 94720, 2005.
- [4] L. Carloni, M. D. Di Benedetto, R. Passerone, A. Pinto, A. Sangiovanni-Vincentelli. Modeling Techniques, Programming Languages and Design Toolsets for Hybrid Systems. Technical report, <http://www.columbus.gr/documents/public/WPHS>, 2002.
- [5] A. Deshpande, A. Gollu, and L. Semenzato. The SHIFT Programming Language for Dynamic Networks of Hybrid Automata. IEEE Transactions on Automatic Control, 43 (4): 584-587, April 1998.
- [6] P. Fritzson, and P. Bunus, Modelica - a general object-oriented language for continuous and discrete-event system modelling and simulation. In Proceedings of the 35th IEEE Annual Simulation Symposium, San Diego, CA, 2002.
- [7] J. Lygeros. Lecture Notes on Hybrid Systems. Rio, Patras, Greece: Internal report, Department of Electrical and Computer Engineering University of Patras, 2004.
- [8] S. E. Mattsson, M. Otter, and H. Elmqvist. Modelica Hybrid Modeling and efficient simulation. In IEEE Conference on Decision and Control, Phoenix, AZ, 1999.
- [9] Modelica Association, Modelica - a unified object-oriented language for physical systems modelling. Language specification. Technical report, <http://www.modelica.org>, 2002.
- [10] M. Otter, K.E. Arzen, I. Dressler. StateGraph - A Modelica Library for Hierarchical State Machines. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, pp.569-578, 2005.

Application of neural networks to model catamaran type powerboats

Garron Fish Mike Dempsey
Claytex Services Ltd
Edmund House, Rugby Road, Leamington Spa, UK
garron.fish@claytex.com

Abstract

Powerboats in operation represent a system consisting of a number of complex components such as: surface propellers, aerodynamics and hydrodynamics; which interact with each other and with the wind and water surface conditions. By measuring the behaviour of the powerboat it is possible to create a mathematical model using system identification methods. A neural network model has been generated which can be used to predict how the powerboat will perform under different driver inputs for the purpose of optimizing performance.

Keywords: neural networks; system identification; powerboats

1 Introduction

There are many different approaches to mathematical modelling and the decision about the most appropriate method to use is based on what *a priori* knowledge is known about the system. Modelica is typically used for white box modelling, which is based on the application of the universal laws and principles. This paper discusses the use of black box modelling techniques that are entirely based on the use of measurement data to generate the mathematical model [1].

In black box modelling, the inputs and outputs of an unknown system are used to create a model that produces an output “close” to that of the actual system, when supplied with the same inputs. Neural network system identification is one method that can be used to create black box models.

In the case of a powerboat, it is convenient to model the system as a black box, as it is not feasible to model the behaviour of the system as a white box model. Figure 1 shows a class 1 powerboat under race conditions. To create a white box model we would need to create models of the aerodynamic and hydrodynamic effects, and their interaction with the

surface propellers and environmental conditions (such as the water surface and wind speed and direction).

The aim of generating a mathematical model of the system was to be able to investigate the effect on the boat performance of variations in driver input and boat setup. A neural network system identification method was selected as the most appropriate way to model the system. Neural network techniques can be very effective at identifying complex nonlinear systems when complete model information cannot be obtained [2].

A Modelica library called ANN_SID has been developed to facilitate system identification using neural networks. The library contains different types of neural network and several training methods and has been applied to study the powerboat system.



Figure 1. View of a powerboat during operation. Image courtesy of Victory Team

2 Neural networks

2.1 An artificial neural network

An artificial neural network is a network of functions called neurons, which are connected by weighted signals (see Figure 2). This architecture is loosely based on a biological neural network. Neural networks can be used for a variety of tasks such as system identification and classification. The ANN_SID library provides neural networks appropriate for system identification tasks.

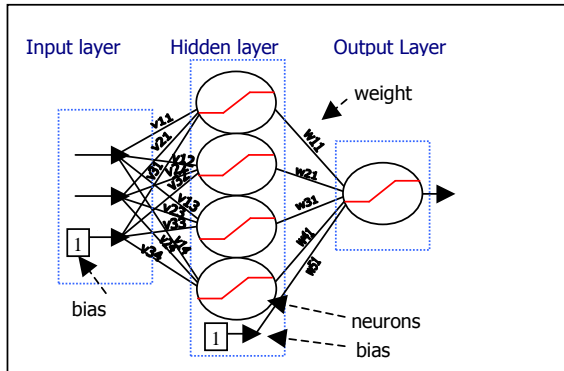


Figure 2. Feedforward neural network

Figure 2 shows a simple diagram of a typical neural network, commonly called a feedforward neural network, which consists of an input layer, a hidden layer and an output layer. In both the hidden and output layers, the weighted sum of the inputs to the layer and the bias, are applied to neuron functions.

The formulae that describe the feedforward neural network in Figure 2 are shown below. Equation (1) calculates the outputs of the hidden layer and equation (2) calculates the output of the neural network.

$$o_j(t) = O_j \left(\sum_i u_i(t)v_{ij} + 1 \cdot v_{n+1j} \right) \quad (1)$$

$$y_k(t) = Y_k \left(\sum_j o_j(t)w_{jk} + 1 \cdot w_{m+1k} \right) \quad (2)$$

where:

o_j is the output of the hidden layer

O_j is the hidden neuron function

u_i is the input

v_{n+1j} is the bias weight for the j neuron (there are n inputs)

y_k is the output of the neural network

Y_k is the output neuron function

w_{m+1k} is the bias weight for the k neuron (there are m hidden neurons)

t is the current sample

Within the ANN_SID library the most common neuron functions such as linear, sigmoid and tanh are available. The user can also easily add their own neuron function by extending from the neuron function base class and implementing the required function.

2.2 Types of neural networks implemented in the ANN_SID library

The ANN_SID library provides pre-defined neural network models for feedforward neural networks and a form of dynamic recursive neural networks called Neural Network Output Error. Within each type of neural network there can be any number of inputs, hidden neurons, neuron layers and output neurons.

In the dynamic recursive neural network, the output of the neural network can be used as an input to the neural network, as shown in Figure 3. This type of recursive network is used for modelling dynamic systems where the next output is affected by the previous output values and previous input values.

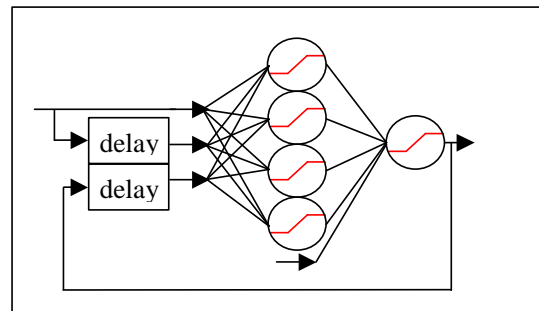


Figure 3. Dynamic recursive neural network

2.3 Training of the neural network

The weights and biases in a neural network have to be trained so that the output of the neural network approximates the actual system well. The mean square error between the actual output and predicted output is the cost function determining the measure of the closeness of the approximation of the neural network to the actual system, as in equation (3).

$$MSE = \frac{1}{2N} \sum_{t=1}^N (z(t) - y(t))^2 \quad (3)$$

where:

MSE is the mean square error

z is the target value (output from the actual system)

y is the output of the neural network

N is the total number of target values

The process of minimising the cost function of the neural network is called training. In the ANN_SID library both backpropagation and the Levenberg-

Marquardt training methods are available. These methods have been implemented in Modelica in both continuous and discrete forms. The choice of method to train a neural network is influenced by the size of the neural network and the amount of data being used to train the network.

The continuous training methods have the advantage that the gradient, which is the rate of change of the weights, is accurate everywhere, not only at the linearization points as with discrete methods. This can result in the search method travelling along the bottom of valleys of the cost function and not oscillating along valley walls.

The continuous method interacts with the variable step solvers to determine the step-size. If the gradient changes suddenly then the solver will reduce the step size to deal with this efficiently. The disadvantage of the continuous method is that it generates huge numbers of equations due to the way that Dymola expands the for loops used in the model. By using Modelica functions and external C functions these problems can be minimized through the reuse of code sections.

Data storage and the manipulation of large matrices in Dymola can also generate problems with large neural networks if the continuous training methods are used. The discrete methods have been implemented to overcome these issues.

2.3.1 Backpropagation

It is possible to train a neural network by calculating the gradient of the cost function with respect to the weights, and to then adjust the weights in the appropriate direction to reduce the cost function. This method is called backpropagation and can be slow to converge to a solution. Appendix A has further information about how the gradient is calculated.

2.3.2 Levenberg-Marquardt

The Levenberg-Marquardt training method generally requires fewer iterations than the backpropagation method to train a neural network. However the LM method is more complex and requires more computation and memory to perform each iteration.

The rules used to calculate the weights are described in Appendix B.

2.3.3 Recursive method

In this method the partial derivative of the neural network with respect to the weights is required. From this partial derivative the gradient and Hessian matrices can be calculated. Once we have determined these matrices either the backpropagation or

Levenberg-Marquardt training methods can be used to minimise the cost function.

Modelica provides semantics to define partial derivatives and Dymola is able to utilise these semantics to generate the symbolic derivative of functions. Example 1 shows how the partial derivatives are defined in Modelica. This method was used to help define a function to calculate the partial derivatives of the neural network with respect to the weights.

[Example: The specific enthalpy can be computed from a Gibbs-function as follows:

```
function Gibbs
input Real p,T;
output Real g;
algorithm
...
end Gibbs;

function Gibbs_T=der(Gibbs, T);

function specificEnthalpy
input Real p,T;
output Real h;
algorithm
h:=Gibbs(p,T)-T*Gibbs_T(p,T);
end specificEnthalpy;
]
```

Example 1. An example of Modelica code for the generation of the partial derivative of a function. Quoted from Modelica 3.0 Specification [4]

2.3.4 ANN_SID Implementation

An example of training a neural network using the ANN_SID library is shown in Figure 4. The training methods are implemented in the replaceable training component and the user simply selects the required method.

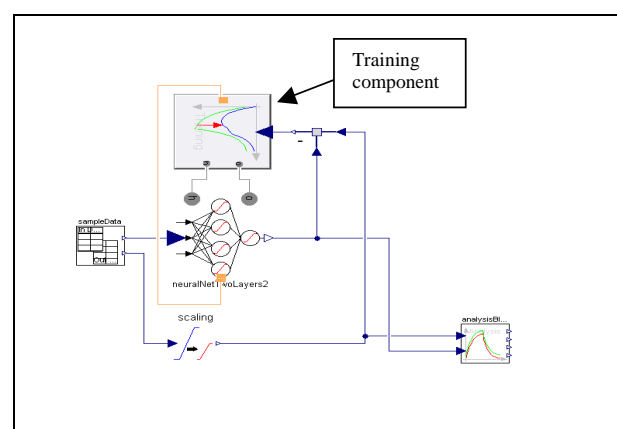


Figure 4. ANN_SID training performed in a model

2.4 Improving the neural network training

When training a neural network it is important to have confidence that the neural network will approximate well with inputs that are not part of the training data. This ability is known as generalisation [5]. One way to investigate this is to divide the data into two sets, one that is used to train the neural network (training data), and one that is used to test the generalisation of the neural network (test data).

Generalisation is likely to be improved by reducing the number of weights used in the neural network [3]. The ANN_SID library supports both weight decay and pruning methods to reduce the number of weights and improve generalisation.

Two pruning methods are available in the library and these are known as Optimal Brain Surgery and Optimal Brain Damage. These algorithms determine which weights to remove from the neural network. The remaining weights are then updated to reduce the errors introduced by removing the weights (for further details refer to [3]).

Weight decay is another approach to removing weights from a neural network. In this method a penalty proportional to the magnitude of the weights is added to the cost function (see [3] for further details). All cost functions should contain a measure of the closeness of the neural network outputs to the desired output. Adding a weight penalty to the cost function generates a trade off between reducing the magnitude of the weights and reducing the closeness measure. As a result of this the weights that have little effect on improving the closeness measure will now be reduced in magnitude.

3 Powerboat operation

The type of powerboat that has been modelled using the ANN_SID library is a Victory Team class 1 off-shore powerboat as shown in Figures 1 and 5. These boats have a catamaran hull with two engines and a central rudder. Each engine drives a height adjustable, steerable propeller. The boats are operated by two crewmembers: a throttle man and a driver. Between them they have 5 controls, which are:

- A steering wheel that directly controls the rudder angle. The steered angle for the propellers is also controlled by the steering wheel angle.

- The propeller heights are set using two rocker switches. These control the trim pistons that move the propellers vertically.
- Throttle position is set using the throttle levers for the left and right engine.

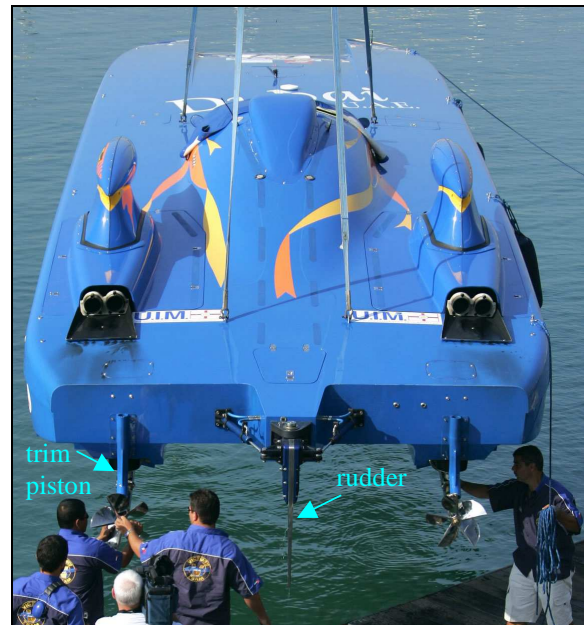


Figure 5. Rear view of a powerboat. Image courtesy of Victory Team

As the boat accelerates it begins to plane and travels higher above the water, i.e. less of the hull is below the water line. As the boat lifts out of the water, the propellers are lowered (trimmed down) to control their depth in the water.

The trim height (or propeller depth) also affects the pitch angle at which the boat travels. In general, the lower the depth of the propellers, the lower the pitch. If the boat is travelling at a pitch angle that is too high for the speed it is doing, it will flip over (a blowover). If the pitch angle of the boat is too low the result will be a larger surface area of the boat in the water and thus an increase in drag.

When cornering, the catamaran powerboat rolls to the inside of the corner (due to the asymmetrical hull design). If the cornering is too severe for the current speed, the boat will begin to roll to the outside of the corner, and will roll over if the drivers do not take correcting action. A typical cornering manoeuvre requires the throttle man to reduce the throttle to slow the boat to a controllable speed before the corner, and the driver to steer the boat along the course, ensuring that the steering angle is not too steep for the current speed.

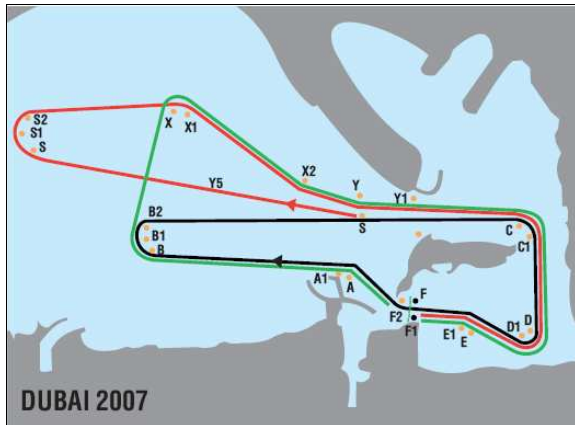


Figure 6. A Racecourse. The powerboats must travel along a course defined by buoys. There are three different types of laps. The start lap in red, the short lap in black and the long lap in green. Supplied courtesy of IOTA.

A race involves the boats travelling around a course defined by buoys laid out in the water (see Figure 6). The drivers try to select good trim height to maximize acceleration while maintaining stability. The drivers also try to find good throttle, rudder and trim positions for cornering that result in fast and stable cornering. The neural network can investigate different possible driver inputs and predict their effects on boat performance over a lap.

4 Powerboat model

4.1 Defining the neural network

To model the powerboat using neural network techniques, a significant amount of data is used to characterise the system. During races and testing sessions the boats are fitted with a data logger that records the data required to train the neural network.

The input data required is:

- The engine throttle positions
- The rudder angle
- The trim height of both propellers

The target output data required is:

- The engine speed of both engines
- The boat speed
- The yaw rate of the boat

Using this input and output data we can train a neural network to represent the powerboat system and then

use the neural network to investigate the system performance with different inputs.

As the boat is an example of a dynamic system, the dynamic recursive neural network was chosen. The model has been generated from data recorded by Victory Team from their racing boat number 77 during the 2007 Arendal race. During this race the boat completed 12 laps of the course.

The measured data was filtered using Basel and Chebyshev filters to reduce the amount of noise and high frequency components in the data. The filtered data was then re-sampled from 100Hz to 1.7Hz to reduce the number of duplicate data points and to decrease the amount of time required to train the neural network. Finally the data was divided into training and test data sets.

4.2 Training the neural network

Training a neural network for such a complex system is done in a number of steps. When first training a dynamic recursive neural network it is not known how many past outputs and inputs will result in the model giving a good representation of the powerboat system. It is also not known how many neurons will be required, or which neuron functions should be used. These can only be determined by trying different configurations to find the best setup.

The first step in training this type of neural network is to train it to only predict the next output value from the previous data value. The weights from this training are then used as the initial weights for the recursive training. The recursive training algorithm described in 2.4.3 was used with the Levenberg-Marquardt method to train the neural network. To improve the generalization, weight decay was used.

4.3 Correlation results

After training, the MSE for the neural network using the training data set was 0.0035 and the MSE for the test data set was 0.0064. This means that the neural network has been trained successfully and is able to accurately predict the performance of the powerboat, as shown in Figure 7.

In Figure 7, the recorded driver inputs have been fed in to the trained neural network and the outputs for boat speed, engine speed and the yaw rate of the boat are compared to the measured data. Overall the results show that there is very good correlation between the neural network and the real powerboat.

There are some small deviations which could be due to a number of different factors, such as swell and

wind conditions along the course, that are not accounted for in the neural network.

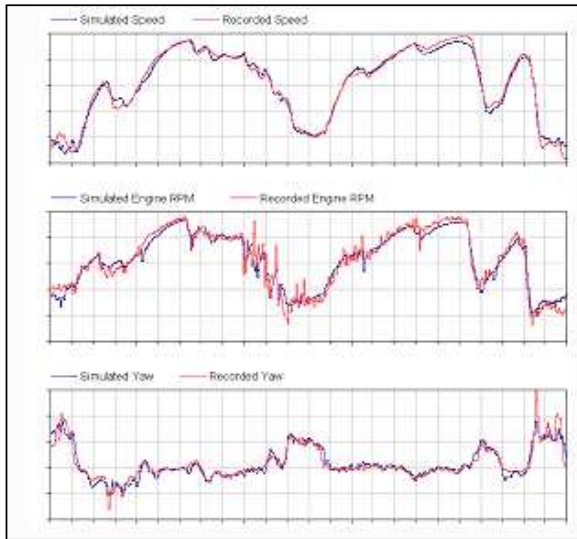


Figure 7. Comparison of simulated neural network with recorded data for a single lap of the course.

4.4 Optimisation of the trim strategy

Section 3 describes how the propeller height (trim height) affects the performance of the powerboat. By using the neural network it is possible to determine what the optimum trimming strategy is for the powerboat.

The model shown in Figure 8 uses the trained neural network to simulate the powerboat accelerating from an initial speed up to its maximum speed.

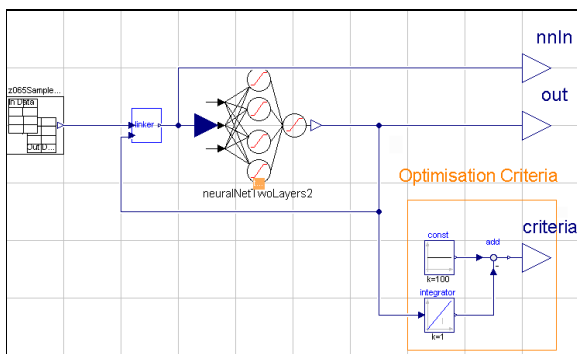


Figure 8. Acceleration model test. The throttle position is set to 100% and the boat is travelling in a straight line.

Figure 9 compares an example trimming strategy extracted from the race data and the optimized trim strategy that has been determined with the use of the

neural network. Using the optimised trimming strategy the powerboat would take 1s less to travel along a 2km straight than using the example trim strategy.

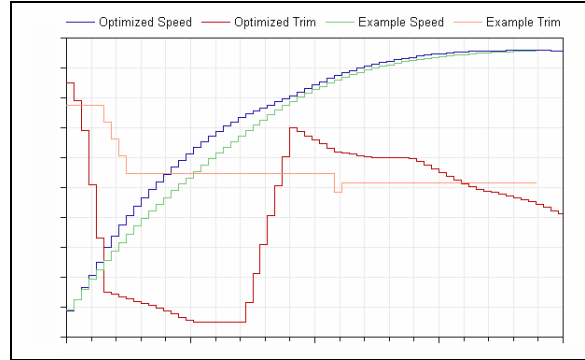


Figure 9. Comparison of a simulated trim strategy with a real trim strategy. The simulated optimal results (Simulated Speed and Simulated Trim) assume the boat is travelling perfectly straight. The Example Trim strategy was taken from the race data and applied to the simulator.

The neural network used in the model can only be expected to accurately model an operating region if this region was sufficiently excited during the data recording stage. In Figure 10 the histogram data identifies what trim position data is available for the operating region of the simulated result. The optimum trim strategy is limited by the availability of data (see Figure 10). The upper bound on the trim data is probably due to driver caution because of the risk of a blowover in this operating region.

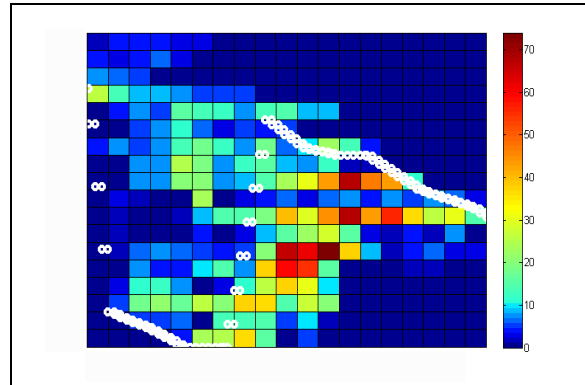


Figure 10. Histogram plot of recorded trim position at the operating state. The optimized trim position is plotted over the histogram as white circles

5 Conclusions

A library called ANN_SID has been developed for the development and training of neural networks for system identification. This library was used to generate a black box model of a powerboat, and this model was then used to determine an improved trimming strategy that should deliver improvements in boat performance.

Acknowledgments

Victory Team have kindly provided the photographs used in this paper and the powerboat data used to carry out this study.

IOTA have kindly provided the course map used in Figure 9.

References

- [1] Estrada-Flores S., et-al. Development and validation of “grey-box” models for refrigeration applications: a review of key concepts. International Journal of Refrigeration, pages 931-946, July 2006.
- [2] Wen Yu, Nonlinear system identification using discrete-time recurrent neural networks with stable learning algorithms. Information Sciences, pages 131-147, 2004.
- [3] M. Nørsgaard, O. Ravn, N.K.Poulsen and L.K.Hansen, Neural Networks for Modelling and Control of Dynamic Systems
- [4] Modelica 3.0 Specifications
http://www.modelica.org/release_of_modelica_3_0/view
- [5] Neural Network FAQ, part 3 of 7,
<ftp://ftp.sas.com/pub/neural/FAQ3.html>

APPENDIX A: Backpropagation algorithm

By calculating the gradient of the cost function (see Section 2.3) it is possible to update the weights in a way that will reduce the cost function. The example below is how backpropagation would be used to up-

date a feedforward neural network using the MSE as the cost function.

The following equations describe a neural network.

$$o_j = O_j \left(\sum_i u_i v_{ij} + 1 \cdot v_{n+1j} \right) \quad (1)$$

$$y_k = Y_k \left(\sum_j o_j w_{jk} + 1 \cdot w_{m+1k} \right) \quad (2)$$

The cost function is the mean square error (i.e. MSE):

$$V = \frac{1}{2N} \sum_{t=1}^N (z(t) - y(t))^2 \quad (3)$$

The calculation of the partial derivative of MSE with respect to output weight w_{jk} follows:

$$\frac{\partial V}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \left(\frac{1}{2N} \sum_{t=1}^N (z(t) - y(t))^2 \right)$$

$$\frac{\partial V}{\partial w_{jk}} = \frac{-1}{N} \sum_{t=1}^N \varepsilon_k(t) \frac{\partial y_k(t)}{\partial w_{jk}} \quad (\text{let } \varepsilon = z(t) - y(t))$$

Substituting in (2):

$$\frac{\partial V}{\partial w_{jk}} = \frac{-1}{N} \sum_{t=1}^N \varepsilon_k(t) \frac{\partial Y_k \left(\sum_j o_j(t) w_{jk} + 1 \cdot w_{m+1k} \right)}{\partial w_{jk}}$$

$$\frac{\partial V}{\partial w_{jk}} = \frac{-1}{N} \sum_{t=1}^N \varepsilon_k(t) \frac{\partial y_k(t)}{\partial a_k} \cdot o_j(t)$$

(where $a_k = \sum_j o_j(t) w_{jk} + 1 \cdot w_{m+1k}$)

The calculation of the partial derivative of the cost function with respect to hidden weight follows:

$$\frac{\partial V}{\partial v_{ij}} = \frac{\partial}{\partial v_{ij}} \left(\frac{1}{2N} \sum_{t=1}^N (z(t) - y(t))^2 \right)$$

$$\frac{\partial V}{\partial v_{ij}} = \frac{-1}{N} \sum_{t=1}^N \varepsilon(t) \frac{\partial y(t)}{\partial v_{ij}}$$

(note that ε and y are vectors)

$$\frac{\partial V}{\partial v_{ij}} = \frac{-1}{N} \sum_{t=1}^N \left(\sum_k \varepsilon_k(t) \frac{\partial y_k(t)}{\partial a_k(t)} \right) \frac{\partial (o_j(t) w_{jk} + 1 \cdot w_{m+1j})}{\partial v_{ij}}$$

$$\frac{\partial V}{\partial v_{ij}} = \frac{-1}{N} \sum_{t=1}^N \left(\sum_k \varepsilon_k(t) \frac{\partial y_k(t)}{\partial a_k(t)} \right) w_i \frac{\partial o_j(t)}{\partial b_{ij}(t)} u_i(t)$$

where: $b_{ij}(t) = O_j \left(\sum_i u_i(t) v_{ij} + 1 \cdot v_{n+1j} \right)$

The discrete weight update method is:

$$\Delta w_{jk} = -\eta \frac{\partial V}{\partial v_{ij}}$$

By choosing η sufficiently small, the cost function can be decreased at each iterate.

The continuous method uses the gradient calculated above to update the existing weights continuously.

APPENDIX B: Levenberg-Marquardt algorithm

In the backpropagation algorithm the search direction is calculated from the first order Talyor approximation of the cost function. The Levenberg-Marquardt algorithm makes use of the second order Talyor approximation of the cost function to update the weights. The second order approximation of the cost function follows:

$$\hat{V}(\theta) = V(\theta^*) + (\theta - \theta^*)V'(\theta^*) + \frac{1}{2}(\theta - \theta^*)V''(\theta - \theta^*)$$

$$\hat{V}(\theta) = V(\theta^*) + (\theta - \theta^*)G + \frac{1}{2}(\theta - \theta^*) H(\theta - \theta^*)$$

where:

θ represents all the weights in the neural network

θ^* are the weights at which the Taylor approximation is made.

V' is $dV/d\theta$ and equal to the gradient G

V'' is $d^2V/d\theta^2$ and equal to the Hessian H

In the Levenberg-Marquardt method a further approximation is made; the Hessian is approximated by the following equation:

$$R(\theta) = \frac{1}{N} \sum_{t=1}^N \frac{dy(t)}{d\theta} \frac{dy(t)}{d\theta}$$

This is valid when the MSE is the cost function.

Let the approximation of the cost function be:

$$L(\theta) = V(\theta^*) + (\theta - \theta^*)G + \frac{1}{2}(\theta - \theta^*) R(\theta - \theta^*) \quad (4)$$

This cost function is minimised using an iterative process; where the next weights are limited to a region around the current weights (see (5)). Limiting the range of the search is often effective as “If the

minimum of L is far from the current iterate, $\theta^{(i)}$, a poor search direction may be obtained.” [3].

$$\theta^{(i+1)} = \arg \min_{\theta} L^{(i)}(\theta)$$

subject to $|\theta^{(i+1)} - \theta^{(i)}| \leq \delta^{(i)} \quad (5)$

where:

$\lambda^{(i)}$ has a monotonic relationship with $\delta^{(i)}$ [3]. Where increasing $\lambda^{(i)}$ decreases $\delta^{(i)}$ and visa versa.

The weights are updated using the following rule:

$$[R(\theta^{(i)}) + \lambda^{(i)}I]\Delta\theta = -G(\theta)$$

where:

$$\Delta\theta = \theta^{(i+1)} - \theta^{(i)}$$

The update rule for the λ value follows:

1. If the $L^{(i)}$ value approximates MSE well, then $\lambda^{(i+1)} = \lambda^{(i)}/2$ and thus increasing the search region.
2. If the $L^{(i)}$ value does not approximates MSE well, then $\lambda^{(i+1)} = \lambda^{(i)*2}$ and thus decreasing the search region.
3. Leave λ^* if neither the 1 or 2 thresholds are true.

To get a more detailed explanation on the update rule for λ^* refer to [3].

ModeGraph -

A Modelica Library for Embedded Control Based on Mode-Automata

Martin Malmheden¹, Hilding Elmqvist¹, Sven Erik Mattsson¹, Dan Henriksson¹, and Martin Otter²

¹Dynasim AB (A Dassault Systèmes Company), Ideon Science Park, SE-223 70 Lund, Sweden

²German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Oberpfaffenhofen, 82234 Weßling, Germany

{Martin.Malmheden, Hilding.Elmqvist, SvenErik.Mattsson, Dan.Henriksson}@3ds.com ,
Martin.Otter@dlr.de

Abstract

The ModeGraph library is a new Modelica library for modeling of hybrid and embedded control systems based on Mode-Automata semantics. Actions can be associated with discrete states in a way that makes sure that the single-assignment rule is fulfilled. Consequently, non-deterministic variable assignment is impossible, which is usual in nearly all other state machine formalisms. Besides Mode-Automata, concepts from Sequential Function Charts (SFC)/Grafcet, Statecharts, and Safe State Machines (SSM) are utilized to provide a flexible modeling environment for safe, hierarchical state machines where Modelica is used as action language. ModeGraph shall replace the existing Modelica.StateGraph library. The implementation of ModeGraph requires extensions to the Modelica language, in order to support the Mode-Automata semantics and to drastically reduce code overhead and improve performance of modeled graphs.

Keywords: Statechart, Mode-Automata, Finite State Machines, Hybrid Control, StateGraph, Modelica

1 Introduction

The StateGraph library [5] is a sublibrary in the Modelica Standard Library 2.1 (from 2004) and later versions, providing components to model hierarchical state machines using Modelica as an action language. The StateGraph library has several significant drawbacks that are mainly due to the underlying implementation language Modelica 2, where some special features needed for hierarchical state machine modeling and for Mode-Automata are missing.

A new Modelica library for modeling hierarchical state machines is proposed in this paper. It is a more Statechart [2] oriented approach compared to StateGraph, but avoids several deficiencies of the State-

chart formalism in order to arrive at safe state machines. The library is capable of handling extended state machine properties, such as hierarchy (meta states), orthogonality (parallel substates), synchronization, and preemption. All StateGraph functionality is available, but with a new simplified implementation. The ModeGraph library ensures safe state machines, especially with respect to

1. upper limit on execution time of one cycle,
2. guaranteed deterministic variable assignment.

The library is based on extensions to the Modelica language, e.g., ensuring mutual exclusivity between states. Usage of the new Modelica 3.0 graphical annotations provides a more modern look and feel.

In the following sections the ModeGraph library will be explained and excerpts of the implementation will be presented. A ModeGraph is defined in Modelica using Boolean equations. As a result, the exact semantics of ModeGraph is formally defined with the Modelica semantics (equations are sorted and iteration takes place, if $\text{pre}(x) \neq x$). General concepts taken from Finite State Machines (FSM), Statecharts [2], Sequential Function Charts (SFC) [7], and Safe State Machines (SSM) [1] will be used as references and benchmarks to demonstrate the feasibility and applicability of ModeGraph.

2 Steps and Transitions

An FSM describes a behavior by decomposing it into a distinct finite set of states visualized by state-transition diagrams. States are usually illustrated by rectangles with rounded corners. An FSM is often used to model reactive systems, which means it reacts to certain stimuli, usually called inputs. A transition is depicted with an arrow between two states and a transition condition written next to the arrow. When the condition evaluates to true, the transition is taken, and a change of state is performed. As an ex-

ample, see Figure 1, where the system initially is in state A. When input α occurs, the state will change from A to B. The arrow originating in a small black dot is used to mark the initial state of the system.

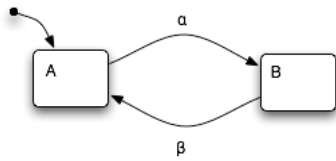


Figure 1: Simple state machine with two states and two transitions.

Inheriting much of the semantics from StateGraph, the basic components of ModeGraph are Steps and Transitions that are both similar to the corresponding StateGraph objects. Figure 2 shows the ModeGraph equivalent of Figure 1. We will proceed to describe the Steps and Transitions in more detail.

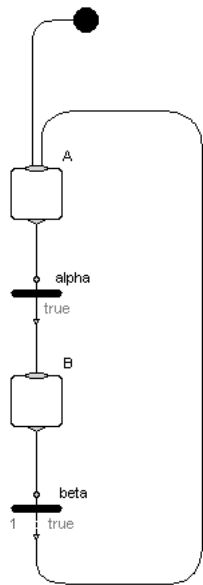


Figure 2: A ModeGraph comprised of two Steps and two Transitions.

2.1 Steps

There are two types of Steps: a regular Step and a StepWithSignal. The state of a regular Step is represented by a Boolean, active. In the case of the StepWithSignal, active is instead a BooleanOutput that can be graphically connected to other components, typically to logical blocks:

```

newActive = (anyTrue(inPort.fire) or
             pre(newActive)) and not
             anyTrue(outPort.fire);
active    = pre(newActive);
  
```

For a Step with one inport and one outport available is defined as:

```
available = active;
```

The function anyTrue iterates through its argument array of connectors and returns true if any of them is true. The state of the Step in the next iteration is called newActive, hence active is set to pre(newActive). A Step is said to be available to the successor Transition when active is true.

Several transitions can lead to and from a Step, respectively. This is implemented with two vectors of connectors, called inPort and outPort. The Step component is said to be a mode, hence only one Step

at each hierarchical level is allowed to be active at a given time instant. This requires restrictions on the outPort fire mechanisms, which will be explained in detail below.

2.2 Transitions

Transitions are used to decide when a change of state should be performed. A basic Transition will check if its predecessor Step is available and evaluate if its transition condition is true (visualised by the condition being colored green). If this is the case, it will send a signal, fire, to its surrounding Steps. Hence, the previous Step will turn inactive and the following will turn active.

```

inPort.fire = condition and
              inPort.available;
outPort.fire = inPort.fire;
  
```

The signal flow between Steps and Transitions is viewed in Figure 3.

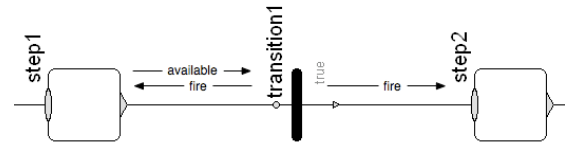


Figure 3: Signal flow between Steps and Transitions.

2.3 Delayed Transitions to Break Loops

Consider the sequence of Steps and Transitions with true conditions in Figure 4. A graph like this is said to be unstable. At a given time instant, the active Step is undefined, because all Transitions will evaluate to true at all times. The code below represents the evaluation of the chain in Figure 4.

```

s1.newActive = (pre(s1.newActive)
                and not t1.fire)
                or t2.fire or entry.fire;
t2.fire = condition and
          pre(s2.newActive);
s2.newActive = (pre(s2.newActive)
                and not t2.fire)
                or t1.fire;
t1.fire = condition and
          pre(s1.newActive);
  
```

Examining this code, it is clear that there is no defined active Step at a given time instant, since it would immediately fire and activate the next Step. Loops like this illustrate the need for a Transition that requires the preceding Step to be available and its condition to be true for a certain period of time before it fires. This is shown by t2 in Figure 5. This type of Transition is called delayed Transition and requires additional equations to decide how long a transition is delayed until it can fire.

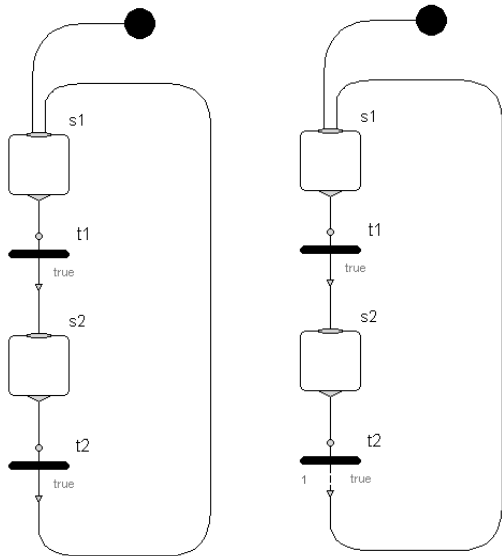


Figure 4: An infinite loop of true transitions. **Figure 5:** A loop broken by a delayed Transition t2.

In the present ModeGraph prototype, a parameter `waitTime > 0` defines the duration for which the fire conditions need to be true before the transition can fire. The release version will alternatively allow definition of the delay by the number of sample periods (with a default of one period), if the ModeGraph is used in a sampled data system. A delayed Transition is currently defined as:

```
enableFire = condition and
                inPort.available;
when enableFire then
    t_start = time;
end when;
fire = enableFire and
        time >= t_start + waitTime;
inPort.fire = fire;
outPort.fire = fire;
```

The concept of delayed transitions is a generalization of the SFC semantics, where every transition from “bottom” to “top” is delayed by one cycle. Introducing delayed transitions explicitly allows drawing state machines arbitrarily without the restriction to always draw it from “top” to “bottom” which is not practical for Statechart-type state machines. Delayed transitions are, e.g., also present in SSM [1], where transitions are by default delayed by one cycle. In SSM “immediate transitions” (denoted with the “#” symbol) are “immediate” and equivalent to the normal Transitions in ModeGraph.

ModeGraph has the essential requirement, that every loop must have at least one delayed transition. In the next section it is described how a violation is detected during translation. This gives both a guarantee that infinite looping is not possible, and it gives an

upper limit on the evaluation time of a ModeGraph at any time instant. Both properties are important for safe embedded control systems.

As mentioned above, Steps can have multiple input and output transitions, and only one Step is allowed to be simultaneously active at every level. This requires priorities among the output transitions. The most intuitive way is to use the index of the port array as priority. A lower index represents higher priority.

The available flag needs to take priority into account and a port is available if the Step is active and if no port with higher priority fires:

```
for i in 1:size(outPort,1) loop
    outPort[i].available =
        if i == 1 then
            active
        else
            active and not
                outPort[i-1].fire;
        end if;
end for;
```

2.4 Graphs with Infinite Loops

Assume that a user creates a graph containing a loop where the conditions of all Transitions are true, as in Figure 4. With the current Step and Transition definitions, the graph will translate, but the solver will not be able to converge towards a single active Step. This kind of undefined behavior is obviously dangerous and is not allowed. To identify cases like this during translation, the signal flow can be slightly changed by introducing a Boolean, `loopTest`. The new signal flow between Steps and Transitions is depicted in Figure 6.

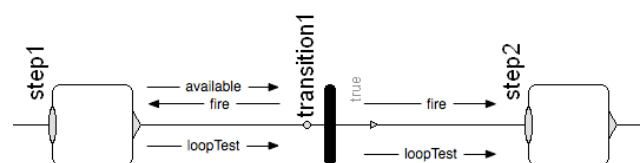


Figure 6: New signal flow with added loop checking.

The idea is to let Steps and undelayed Transitions just pass the signal on, while a delayed Transition and all entry points will set `loopTest` to true. If only Steps and undelayed Transitions are present in a loop, the translator will recognize an algebraic loop of Boolean equations, and will print an error message because Boolean algebraic loops cannot be solved. If a delayed Transition is included, the algebraic loop will be broken, and the graph will safely translate. The code for the loop testing is simple:

In a Step:

```

for i in 1:size(outPort,1) loop
  outPort[i].loopTest =
    anyTrue(inPort.loopTest);
end for;

```

In a Transition:

```

outPort.loopTest = inPort.loopTest;

```

In a delayed Transition

```

outPort.loopTest = true;

```

This “brute force” method has the slight drawback that no better loop breaking check can be provided. In principal, it might be possible to have only undelayed transitions and if the transition conditions are restricted, it might be possible to prove that infinite looping is not possible.

3 Encapsulation and Aggregation

The FSM formalism is adequate as long as the modeled behavior remains reasonably simple. When the number of states and transitions increases, the complexity of the FSM grows exponentially. This is fatal to readability and strongly confines the viability of the graph. Thus, when a state machine grows in complexity, a strong formalism should support object-orientation and proper encapsulation of isolated parts of the behavior to ensure well-defined interfaces.

Some remedies for the mentioned problems were introduced by David Harel in Statecharts [2], where several new properties were presented to extend FSM. Being able to cluster states into a superstate makes it possible to identify similarities between a number of states and draw advantages from common properties among them. Clustering of states enables reuse of larger parts of a behavior than just a single state. The superstate has a default entry point, which is connected to the initial state with the same notation as the initial state arrow. In Figure 7, B and C share the common property of transition β leading to state A.

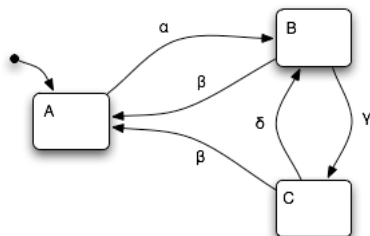


Figure 7: Three states of which two share common properties.

Thus, B and C can be clustered together into state D in Figure 8. Note the improved visual appearance in

Figure 8 compared to Figure 7, despite the exact same behavior of states A, B, and C.

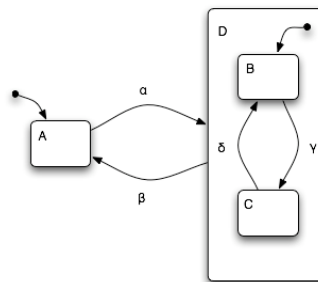


Figure 8: Two states clustered together in a superstate.

Refinement of a state involves identification of a number of child states with unique properties within a particular state. In Figure 8, states B and C can be said to be a refinement of state D. Hence, state D is said to be the superstate of state B and C. Being in one of the substates implicitly means also being in the superstate. The superstate D in Figure 8 is said to be the XOR-decomposition of its substates.

3.1 ModeGraph Composite

ModeGraph allows aggregation of states into superstates. A Composite component inherits from ModeGraph.Composite and has inPort and outPort connectors defined, like a regular Step, but also suspend ports and resume ports - like in StateGraph. Figure 9 shows a ModeGraph corresponding to the chart in Figure 8.

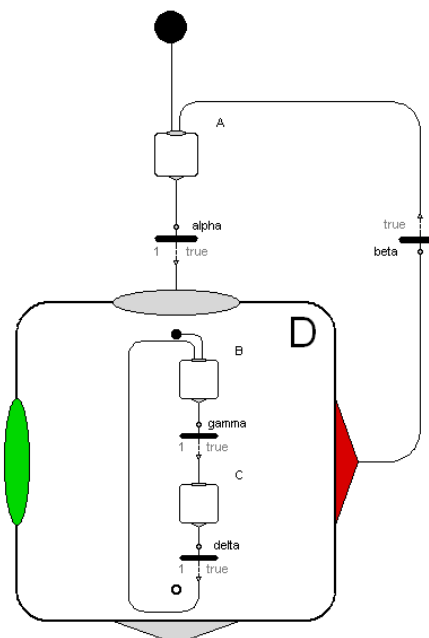


Figure 9: ModeGraph containing two Steps clustered inside a Composite. State D is a Modelica mode block where the diagram layer is visible in the icon. Compare with Figure 8.

The initial Step, B, of the Composite is connected to the entry port, depicted with a black dot. Similarly, there is an optional exit port, illustrated with two circles at the bottom of the Composite. This notation is inspired by the semantics of SSM, but is slightly modified to provide a more consistent look. In SSM, a specific 'final step' indicates when the superstate may be exited through the outPort, and is depicted with two circles. To prevent misuse, there is an exit port in the Composite and Parallel ModeGraph components that the 'final step' should be connected to. When this step is active, the outPort of the Composite becomes available.

The difference between entry/exit and the existing StateGraph approach extends beyond the mere graphical deviation. The entry model contains a state connected to the black connector dot that is initially true. Having an entry state, no specific InitialStep component is required. This prevents the user from making mistakes by, for example, placing two InitialStep components in a graph. The code below defining the entry point ensures that the state remains true for one iteration, when the Composite turns active, and then switches to false.

```

    Entry entry(fire(start = false,
                    fixed = true));
protected
    Boolean active(start = true;
                 fixed = true);
equation
    active = pre(active) and not
            pre(entry.fire);
    entry.fire = pre(active);
    
```

When the Step connected to the exit port is active, the Transition connected to the outPort of the Composite may fire (if its condition is fulfilled). This calls for a definition of how the state of a Composite is evaluated:

```

    available = exit.exit.available and
                allSubBlocksFinished and active;
    newActive = (active and not
                anyTrue(outPort.fire) and not
                anyTrue(suspend.fire)) or
                anyTrue(inPort.fire) or
                anyTrue(resume.fire);
    active = pre(newActive);
    
```

In the code above, the state of the Composite, `active` is set to `pre(newActive)` to avoid an algebraic loop involving mode conditions that will be introduced later in this paper.

An important feature of ModeGraph is conditional execution. This applies for the Composite component, whose associated code is only executed when the composite is active. This will be further explained in Section 6.

4 Preemption and Exception

Aggregation of states introduces new possibilities. Being an own entity, it is possible to have a transition drawn directly from the superstate. This will result in a preemption, and the superstate is left regardless of which of the substates is active, see, e.g., transition β in Figure 8. Of course, normal exit is possible by having a transition originating in an inner state and targeting an outer. Notice how state D in Figure 10 is only left through transition β if state C is active.

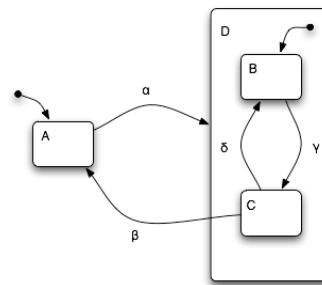


Figure 10: Superstate D can only be left when in substate C.

4.1 ModeGraph Exit and Preemption

To exit a Composite, the final step is connected to the mentioned exit port. When the final step is active, `exit.exit.available = true`, and a transition connected to the Composite outport becomes enabled. The ModeGraph realization of Figure 10 is shown in Figure 11.

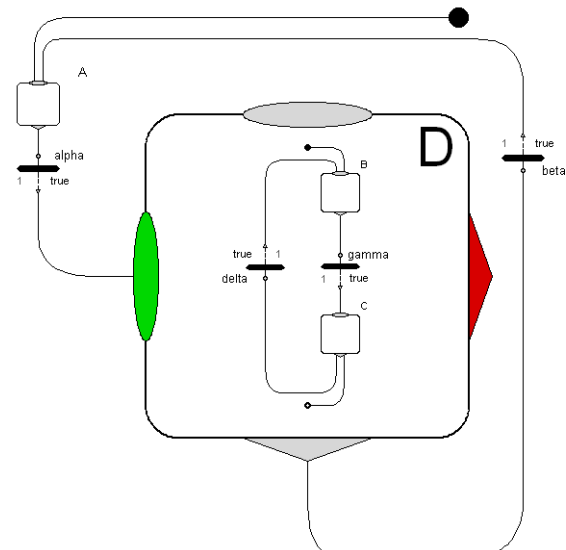


Figure 11: Composite D can only be left if Step C is active, compare with Figure 10.

A ModeGraph Composite has an array of suspend connectors. Recalling the active condition of the Composite, it is clear that after a suspend port fires,

the Composite is no longer active. This behavior is used to preempt a Composite without necessarily having reached the final Step, i.e., the one connected to the exit port. The condition of `suspend.available` needs to equal the state of the Composite, since it should be preempted only when it is active. The same kind of prioritization as for Steps is performed here:

```

for i in 1:nSuspend loop
  suspend[i].available =
    if i == 1 then
      active
    else
      active and not
        suspend[i-1].fire;
    end for;
end for;

```

The suspend port can be compared to the Statechart equivalence of drawing a transition directly from the superstate to an outer state, compare for example transition β in Figure 8 and its equivalent in Figure 9. The deactivation of the Composite does not, explicitly, influence the internal states of the Composite. The state of the subblocks will be kept, but all internal interaction will be frozen.

4.2 History and CLH

The concept of preemption introduces an additional way of entering a superstate. Normally, entry is performed through the default entry point, as mentioned above. This behavior can be compared to a subroutine that has only one entry point. There is an obvious advantage of offering additional ways of entering an aggregation, similarly to the ways a co-routine may be entered. Hence, re-entering a superstate, it is also reasonable to be able to enter the most recently visited substate.

Memory of the internal state of a superstate is called “entry by history” in Statecharts, and depicted with an encircled H to which transitions can be connected. The H-entry will make the previously visited state before preemption at the current level active. If the superstate is entered for the first time, the default entry arrow is used. Assume for example that state C is active and transition β is taken in Figure 12. If subsequently transition α is taken, state C (and of course also state D) will once again be entered.

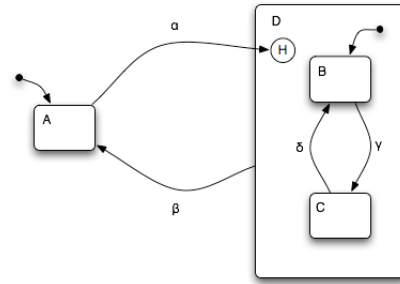


Figure 12: Superstate D is entered through an H-entry.

To handle history of several nested superstates, the H-entry can be extended to be applied all the way to the lowest level. This is in Statecharts called an H*-entry. Assume that state C in Figure 13 is active, and transition β is taken (leaving superstate F). If later transition α is taken, state C will be active, since α is connected to an H*-entry.

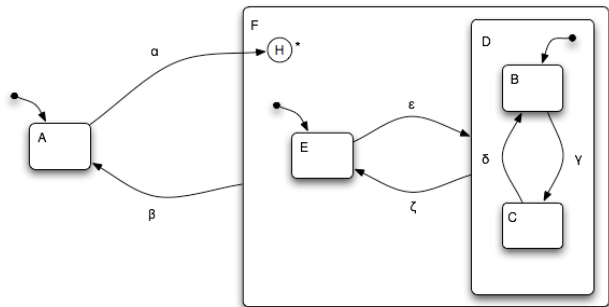


Figure 13: Superstate D is entered through an H*-entry.

Having the possibility to utilize history functionality, an obvious requirement is to also clear this memory and enter an aggregation as normal. We will introduce the concept of actions and activities before this property is defined.

4.3 Actions and Activities

A transition action in FSM can be performed when a transition fires, which is denoted at the transition condition after a '/' character. An action is assumed to be performed instantaneously in ideally zero time.

Statecharts also defines activities that, opposed to actions, are performed in non-zero time, and are used to carry out tasks of some sort. For each activity ∂ , the following two actions are defined: `start(∂)` and `stop(∂)` which are true when an activity starts and stops, respectively. Also, a new condition is defined: `active(∂)`, which is true when ∂ is active.

In SFC, actions are associated with a state instead of being executed upon a transition being fired. Actions in SFC are not instantaneous as in Statecharts and may also be conditional.

4.4 CLH

With the definition above, a special action called clear-history(state), $clh(state)$, can now be defined. When clh is performed, the history at the level of the state is reset. Just as with the H-entry, it is possible to perform a clear-history down to the deepest level. This action is consequently called $clh(state^*)$. Consider the graph in Figure 14 and assume that state C is active when transition β is taken and $clh(F)$ is performed.

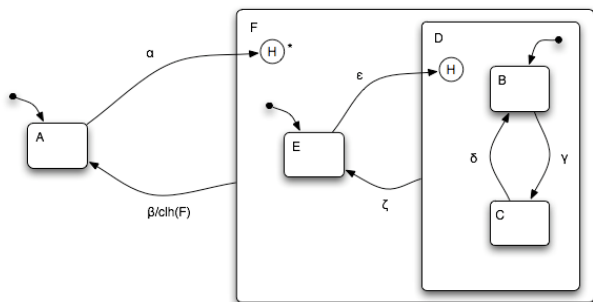


Figure 14: The history of superstate F is reset when transition β is taken.

If transition α is subsequently taken, the choice stands between state D or E, and since $clh(F)$ has been performed at this level, the default arrow, and consequently, state E will be active. Note that if now transition ϵ is taken, state C will be active, since no clh occurred at this level.

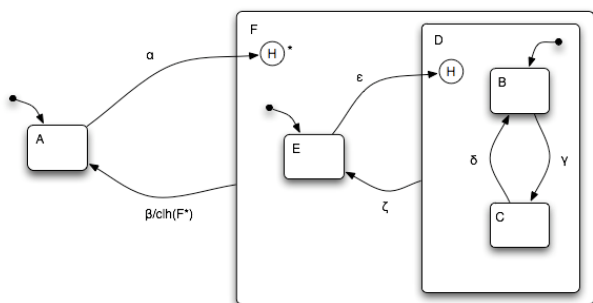


Figure 15: The history of superstate F and all descending substates are reset when transition β is taken.

In Figure 15 $clh(F^*)$ is performed instead. If now transition α is taken, state E would be entered. If transition ϵ is taken, it would result in state B being active, since all superstates are entered through their respective default arrows on all descending levels due to the earlier performed recursive clh .

4.5 ModeGraph History and CLH

The ModeGraph equivalence of the History junction is the resume port. When the resume port fires, the Composite is simply activated. This means that a superstate that is always entered through a history

junction, like the one in Figure 12, is directly implementable in ModeGraph by always entering through the resume port, like in Figure 16.

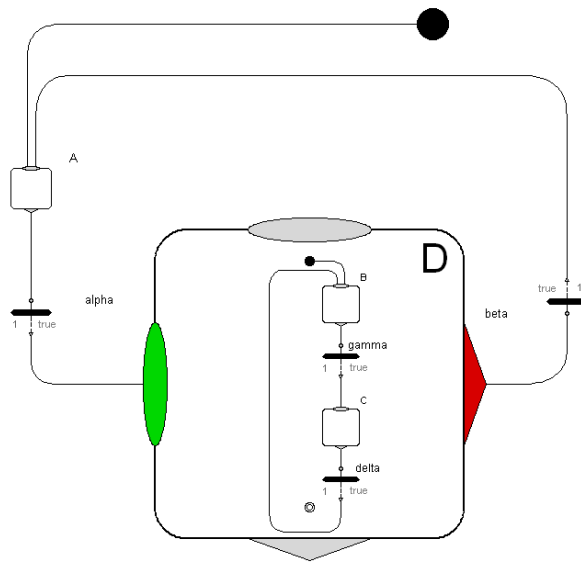


Figure 16: ModeGraph Composite being entered only through the resume port, compare with Figure 12.

Note that when a Composite is suspended, all states all the way down the hierarchy keep their current state, which actually corresponds to the H*-entry. Figure 17 is the ModeGraph implementation of Figure 13. Clear History is performed in ModeGraph upon normal entry through the inPort of a Composite.

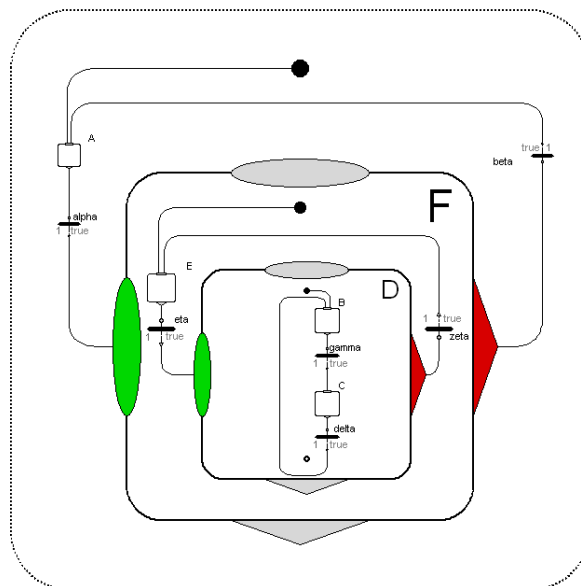


Figure 17: Two nested ModeGraph Composites that are both entered through their resume ports, compare with Figure 13.

5 Parallelism

Parallelism and synchronization are important properties of a state machine to prevent exponential blow-up of the number of states as complexity grows. Assume, for example, two subsystems having x and y states, respectively. When executing in parallel, the number of states would obviously be $x + y$. However, realizing the system without the parallel states would require $x \cdot y$ states.

Orthogonality provides the possibility to have several superstates executing in parallel. Assume state D being the orthogonal product of states B and C , $D = B \times C$, then D is said to be the AND-decomposition of B and C , see Figure 18.

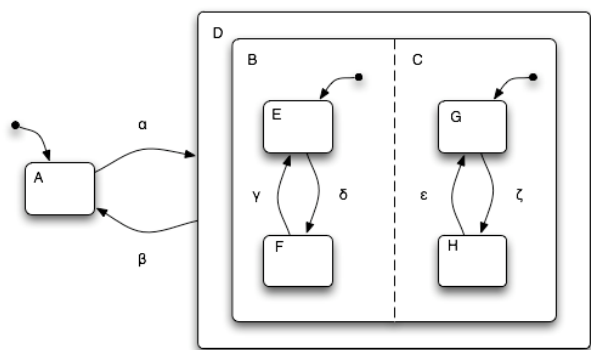


Figure 18: Superstate D is the orthogonal product of B and C .

In practice, it is common to graphically omit the surrounding orthogonal product state, and in this case instead connect transitions directly to the $B \times C$ state.

Another important aspect of subsystems running in parallel is synchronization. An orthogonal product of states should provide the possibility of only being left if a particular set of states is active. In Statecharts, this is performed by using guards on a preemptive transition originating in the orthogonal product state. This can successfully be used to let sequences synchronize before continuing further execution.

Being a sequence-control-oriented formalism, SFC/Grafcet implements parallelism somewhat differently compared to the illustrated example. In SFC, a transition can be split up in parallel paths. Consequently, several paths can be joined by an AND-junction. This sequential approach suits its sequence control purposes very well, and supports synchronization in a natural way.

5.1 ModeGraph Parallelism

The existing StateGraph Parallel component follows the Grafcet/SFC tradition by dividing one connection

into a new given number of subpaths that are later joined to a single connection. Hence, synchronization is implicitly demanded of parallel branches. However, it is sometimes useful to have subsystems working independently of each other that never synchronize, as is the case for states B and C in Figure 18. Those two systems will run concurrently until they are preempted by transition β . Hence, no synchronization will ever occur in this case.

Implementing this in StateGraph will result in a rather messy graph with an unconnected Parallel join component, see Figure 19. This use of unsynchronized subsystems is common in Statecharts, and a more flexible way of implementing orthogonality is thus desirable.

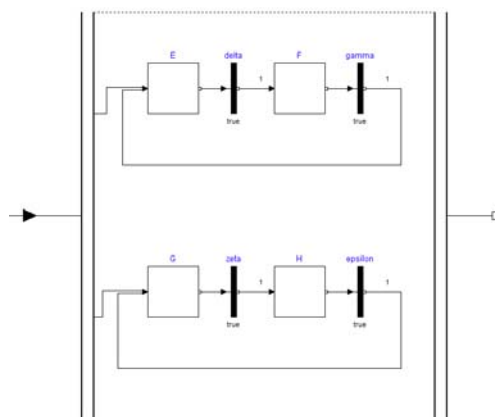


Figure 19: A StateGraph containing two unsynchronized subsystems.

In ModeGraph, a more Statechart-oriented design is introduced without compromising existing possibilities of synchronization. A Parallel component inherits from ModeGraph.Parallel and is placed within a Composite to enable preemption. Figure 20 shows a ModeGraph implementation of Figure 18.

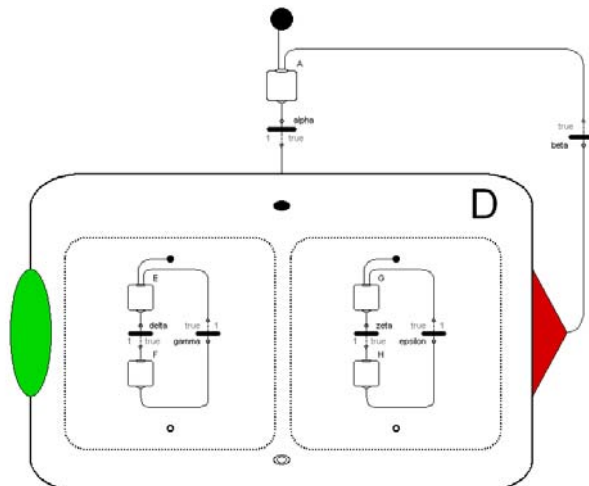


Figure 20: A ModeGraph Composite that contains two independent Parallel subsystems.

As can be seen, ModeGraph incorporates an approach to orthogonality that is very similar to Statecharts. Note, that one or more Parallels are placed in a Composite to provide the possibility of preemption and synchronization of the Parallel children. As shown in the code below, the active flag of a Parallel component is always true. The reason for this is that its activeness should always be decided by the parent Composite. Alternatively, if the Parallel is the root of the graph, it should indeed always be active.

```

output Boolean active
    "= true if parallel step is
    active, otherwise the
    parallel step is not active";
equation
    active = true;
    
```

One important feature of the ModeGraph Parallel component is that synchronization is still available. Each Parallel block also contains a Boolean variable, `finished`, which is true when the Step connected to the exit port is active.

Assume the scenario in Figure 18 with the modification that transition β can be taken only if Step F and Step H are simultaneously active. This would result in a ModeGraph implementation shown in Figure 21.

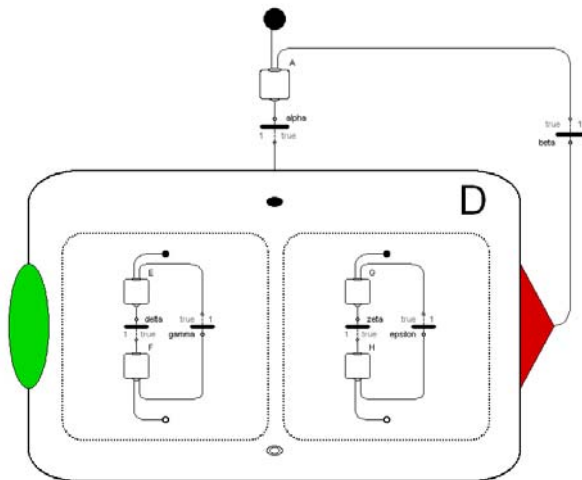


Figure 21: A ModeGraph Composite with two parallel subsystems that must synchronize to allow the Composite to exit. Note that the exit ports of the Parallel components are now connected.

To utilize exit connectors of the Parallel component, it is required to set the parameter `withExit` to true. If `withExit` is false, `finished` will be set to true. This becomes useful when synchronizing Parallel states with exits when there are additional Parallel states without exits present in the same Composite.

The new approach of parallelism supports safe graphs in a natural way. As stated in [5] the Parallel and Alternative components in StateGraph are vulnerable to misuse. The problem is that the Alterna-

tive/Parallel components are instantiated at the same level as their branches. This makes it possible for a user to freely connect a branch outside the component without properly synchronizing it, see Figure 22 for an example. Analysis to identify such cases forces unnecessary code overhead.

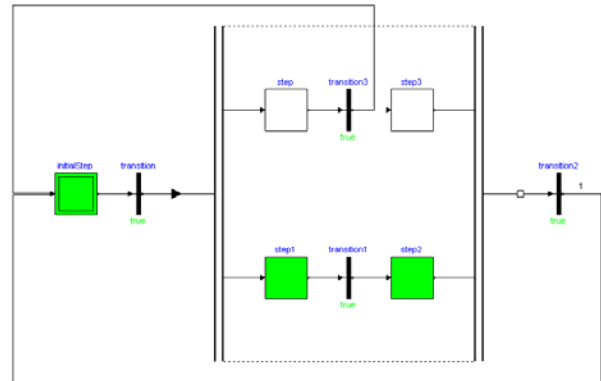


Figure 22: Example of unsafe StateGraph.

In ModeGraph, this kind of misuse is not possible. Since the user is forced to inherit from `ModeGraph.Parallel` and build the parallel branches within a model, i.e., on a different level, there is no way of connecting to outer Steps or Transitions, since the icon layer is closed.

6 Modelica Mode

To implement Mode-Automata in Modelica, a mechanism for enabling/disabling a block is needed. There must be a way to conditionally evaluate code within a Composite and enable/disable its children. The Modelica mode comprises five variables that define the behavior of the inheriting block. The variables define under what conditions equations within the block and its children will be evaluated and when to reset states and outputs. The proposed built-in base class mode is defined as:

```

partial block mode
    input Boolean finished = false
        "The execution of the mode
        block is finished";
    protected
        Boolean enable = true
            "Enable/disable block and all
            children";
        Boolean enableSubBlocks = true
            "Enable/disable children";
        Boolean resetStates = false
            "Reset all continuous and
            discrete states of this block
            and all its children";
    
```

```

Boolean resetOutputs = false
  "When a block is disabled, set
   all its outputs to their start
   values";
end mode;

```

The translator will assert that only one block inheriting from mode at every level is enabled at the same time instant. This will make it possible to ensure consistency of the single assignment rule in the Mode-Automata context.

Naturally, the ModeGraph Step component extends mode, and only one of Steps A and B in Figure 2 can thus be enabled at a given time instant. A proposed Modelica extension would make it possible to assign a variable *y* as:

```

Step A equation
  y = expr1;
end equation;
Step B equation
  y = expr2;
end equation;

```

The same restriction as in a when-clause applies, i.e., there must be a variable reference on the left hand side of the equal sign (here: *y*). This code will be transformed by the translator and will result in the following single equation:

```

y = if A.enable or
    A.enableSubBlocks then
    expr1
    elseif B.enable or
    B.enableSubBlocks then
    expr2
    else pre(y);

```

The expressions *expr1* and *expr2* are thus defined within A and B, respectively, and the equation above is generated by the translator to ensure that the single assignment rule is not violated.

As a consequence this means that in the generated code every variable is only defined at one place. For example, it will not be possible to assign the same variable in two parallel branches of a Composite step with two Parallel modes. If this is attempted, an error occurs, since the number of equations and unknowns is not the same. Nearly all other formalisms lack such a property and therefore it is possible to assign to the same variable several times and then non-intuitive rules are used to determine which assignment takes priority. Stated differently, ModeGraph guarantees deterministic variable assignment, whereas most other state machine formalisms have non-deterministic variable assignment.

6.1 Composite Mode

Just like the Step, the Composite component inherits from the mode base class. It is by purpose that a Composite and a Step on the same level are mutually exclusive. All components inside the Composite will in turn be gathered and evaluated in the same manner.

The modifiers of the mode block need to be configured according to the desired behavior of the Composite. When the inPort fires, *resetStates* is set to re-initialise all the states of the Composite and its children to behave exactly like if it was indeed the first time it was entered. The attribute *enableSubBlocks* will be true when the Composite is active, enabling children as long as the Composite stays active. When the block is not enabled, all outputs of the Composite and all children should be reset, hence *resetOutputs* is set to true. The mode modifier is shown below.

```

partial block Composite
  extends mode(
    enableSubBlocks = active,
    enable = true,
    resetStates = inport_fire,
    resetOutputs = true,
    finished = allSubBlocksFinished);

```

The proposed built-in operator *allSubBlocksFinished* expands to a check if all children of the mode have their finished variable set to true. Hence, if *allSubBlocksFinished* is true, the Composite may be left through the outPort, since its finished flag becomes true.

6.2 Parallel Mode

The final discussion relates to the Parallel Component. Since we think in terms of an orthogonal product, $A \times B$, several Parallel components will indeed be simultaneously active. To avoid violation of the Mode-Automata semantics, the Parallel component is not itself a mode, but contains sets of modes. Since the sub-components are not instantiated at the same level as the Parallel components this does not conflict with the Mode-Automata theory.

Since the Parallel component is not a mode, it is not conditional. There is, however, no need for this, since Parallels are placed inside Composites, and thus 'inherit' the conditional behavior of the parent Composite. Note that a Parallel can be placed at the top level. In fact, this is the intended way to define a top level ModeGraph, since the top component of a graph should always be active.

7 Application Example – Harel’s wristwatch

When David Harel introduced Statecharts in [2], he identified and mapped the behavior of a Citizen Quartz Multi-Alarm III wristwatch using the new semantics. This complex, yet comprehensible graph has been realized in ModeGraph as a case study. Selected parts of the ModeGraph implementation of Harel’s wristwatch [2] will be used to illustrate the functionality of the mode concept. The main interface of the ModeGraph implementation is shown in Figure 23. It is comprised of a main display, buttons for interaction, and indicator lamps to show the status of the alarms.

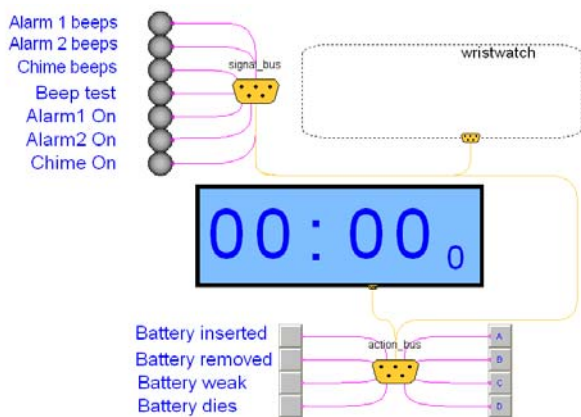


Figure 23: ModeGraph Wristwatch main window.

More information about this implementation of Harel’s wristwatch can be found in [3].

An example where the mode semantics becomes very convenient can be found in the time update mechanism of Harel’s wristwatch. In update mode, different time quantities can be traversed by pressing a button, c. When another button, d, is subsequently pressed, the quantity defined by the active state is incremented, see Figure 24.

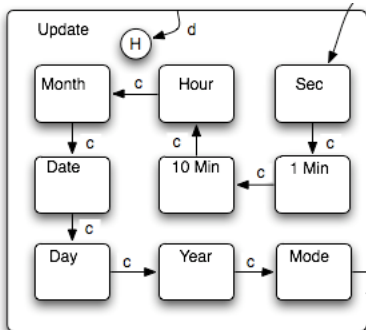


Figure 24: Update mechanism of wristwatch.

The ModeGraph realization of Update is shown in Figure 25.

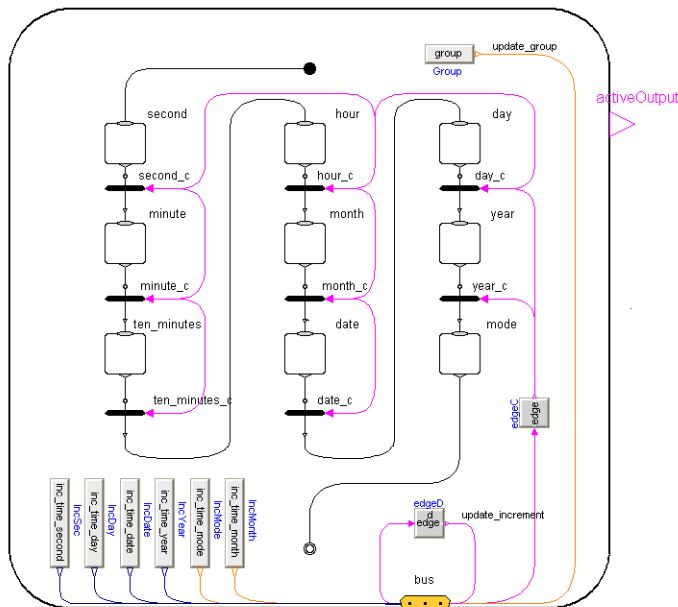


Figure 25: ModeGraph realization of Update.

Declaration of the Step components should according to the proposed mode declaration look like:

```

Step second equation
  inc_time_second = 1;
end equation;

Step minute equation
  inc_time_second = 60;
end equation;
    
```

```

Step day equation
  inc_time_day = 1;
end equation;
    
```

Hence, `inc_time_second` would, e.g., be automatically gathered into a single if-statement like:

```

inc_time_second =
  if second.enable
    or second.enableSubBlocks then
    1
  elseif minute.enable
    or minute.enableSubBlocks then
    60
  ...
  else pre(inc_time_second);
  ...
    
```

Harel’s wristwatch contains a state, `chime-status`, shown in Figure 26. This state controls the chime function that is an alarm that sounds every whole hour that may be either enabled or disabled. Additionally, when enabled, it can be either quiet (the default) or beeping every time the clock reaches a whole hour. Notice that when `chime-status` is active, it can be left regardless of which of the internal states is active. The ModeGraph realization of `chime-status` is shown in Figure 27. Recall that every time a ModeGraph Composite turns inactive,

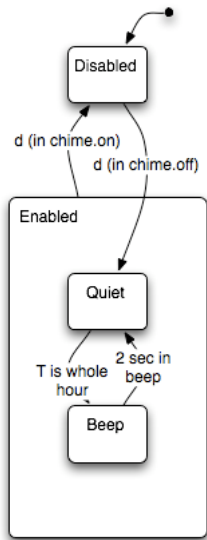


Figure 26: Chime – Status.

interactions between all child states are frozen and no code within the block is evaluated. Hence, if the Composite is activated anew, the last active sub-blocks will once again be active. When entering state enabled, sub-state quiet should be activated by default. In the ModeGraph realization, the step representing state quiet is connected to the entry point. Hence, entering enabled through the inport, resetStates becomes true, and the Step connected to the entry point will be active.

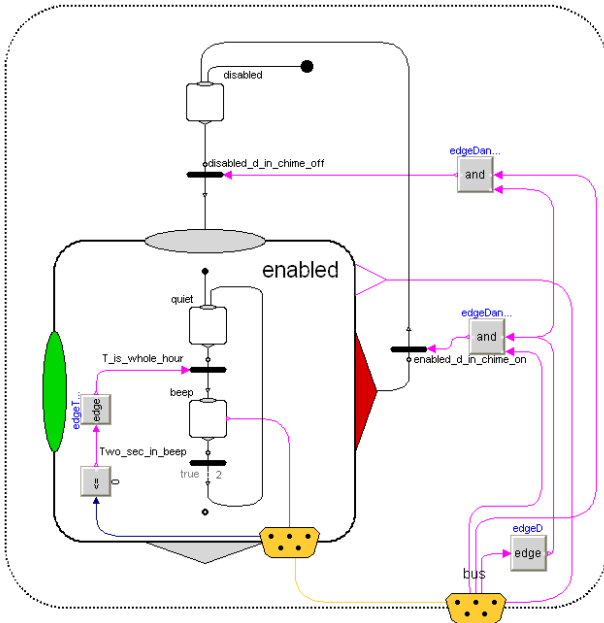


Figure 27: ModeGraph realization of Chime-Status.

The state stopwatch in Harel’s wristwatch is a good example of the need of flexible parallel states that support easy synchronization. The Stopwatch can either display zeros or the running/frozen time, depending on the context of the parallel states display and run, see Figure 28.

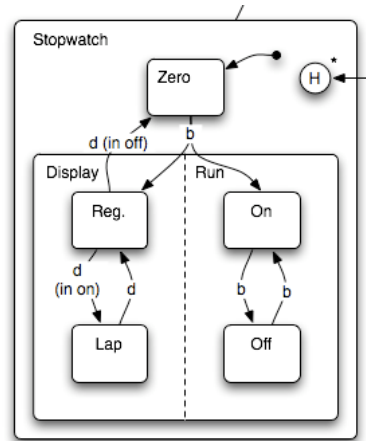


Figure 28: Stopwatch.

When the states display and run are entered, the stopwatch starts running (state on) and displays regular time (state reg). Pressing button b, the user can turn the stopwatch on/off. Pressing button d has different meanings depending on the current active state of run. If the stopwatch is running, pressing button d switches between display modes regular and lap. If instead the stopwatch is in state off, button d is used to exit to state zero, thus resetting the time of the stopwatch. The ModeGraph realization of the stopwatch is shown in Figure 29.

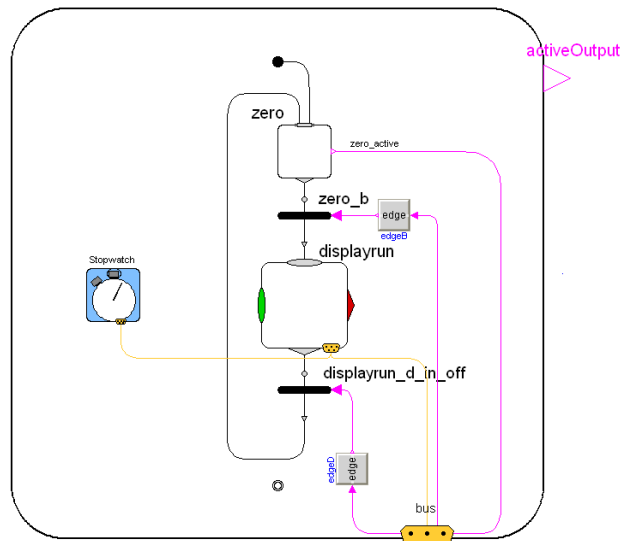


Figure 29: ModeGraph realization of Stopwatch.

An additional Composite displayrun is introduced to encapsulate the two parallel states display and run, see Figure 30. The transition condition $d(in on)$ in Figure 28 becomes true when button d is pressed and state on is active. It is realized in ModeGraph by the state on (in Parallel run) sending its state out on the bus, which is read by the transition $reg_d_in_on$ located in the Parallel display. This is a good example of how inter-mode

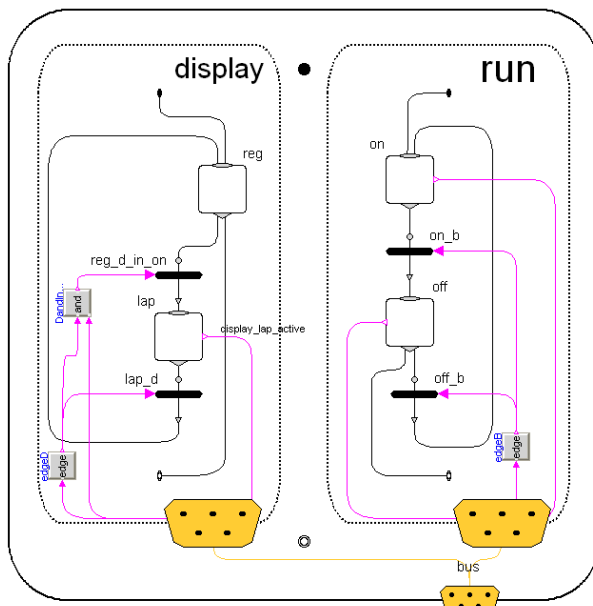


Figure 30: Contents of the ModeGraph Composite displayrun.

communication can be performed with expandable connectors, often called buses. This is an important difference in ModeGraph compared to other types of state machines. Since ModeGraph is implemented in Modelica and modes are basic blocks, the variables in a mode block are local variables. In other formalisms, variables are usually available as global entities on all levels. For embedded systems the ModeGraph approach is safer, since variables of composites are encapsulated.

Also note, in Figure 30, how Steps `off` and `reg` are connected to the exit points of their respective Parallel parent. When both these Steps are active, Parallels `run` and `display` both declare themselves finished, which enables transition `displayrun_d_in_off` in Figure 29 to fire, since its `allSubBlocksFinished` attribute will return true.

What has just been discussed is the core functionality of the ModeGraph library. The possibility of simply ignoring equations within a disabled mode, that also are guaranteed to be mutually exclusive with respect to other modes on the same level, reduces code and introduces powerful properties allowing equations to be associated with modes.

8 Conclusions

In this paper the ModeGraph library has been introduced. The motivation for ModeGraph originates in the inadequacy of StateGraph in terms of implementing Statechart-oriented state machines. ModeGraph offers improved flexibility of graphical modelling of state machines, regardless if they are SFC/Grafcet-

or Statechart-oriented. Graphically, ModeGraph provides a modern look and feel with components based on Modelica 3.0 graphical annotations. Furthermore, the Mode-Automata semantics offers a convenient way of managing complex conditional structures for the user. Large-scale systems will successfully draw advantage of the fact that only relevant parts of the code (i.e., the code of the current active modes) are evaluated. The conditional structure also prevents the user from unintentionally abusing the available components in dangerous ways without having extensive code overhead.

9 Acknowledgement

This work was in parts supported by the ITEA2 EUROSYS LIB project

(http://www.itea2.org/public/project_leaflets/EUROSYS LIB_profile_oct-07.pdf).

References

- [1] André, C. (2003): **Semantics of S.S.M (Safe State Machine)**. I3S Laboratory – UMR 6070 University of Nice-Sophia Antipolis / CNRS. <http://www.i3s.unice.fr/~map/WEBSPO RTS/Docu ments/2003a2005/SSMsemantics.pdf>
- [2] Harel, D. (1987): **Statecharts: A Visual Formalism for Complex Systems**. Science of Computer Programming 8, 231-274. Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel. http://www.inf.ed.ac.uk/teaching/courses/seoc1/20 05_2006/resources/statecharts.pdf
- [3] Malmheden, M. (2007): **ModeGraph – A Mode-Automata-Based Modelica Library for Embedded Control**. Master's thesis, Department of Automatic Control, Lund University, Sweden. <http://www.control.lth.se/database/publications/arti cle.pike?artkey=5808>
- [4] Maraninchi, F. and Rémond, Y. (2002): **Mode-Automata: a New Domain-Specific Construct for the Development of Safe Critical Systems**. <http://www-verimag.imag.fr/~maraninx/SCP2002.html>
- [5] Otter, M., Årzén, K.-E., Dressler, I. (2005): **State-Graph - A Modelica Library for Hierarchical State Machines**. Proceedings of the 4th International Modelica Conference. TU-Hamburg-Harburg, Germany. http://www.modelica.org/events/Conference2005/o nline_proceedings/Session7/Session7b2.pdf
- [6] AFCET. (1997): **Normalisation de la représentation du cahier des charges d'un automatisme logique**. J. Automatique et Informatique Industrielle.
- [7] IEC Standard 61131-1 <http://www.iec.ch/>

A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment¹

Ulrich Donath Jürgen Haufe
Fraunhofer-Institute for Integrated Circuits, Design Automation Division
Zeunerstraße 38, 01069 Dresden, Germany
ulrich.donath@eas.iis.fraunhofer.de juergen.haufe@eas.iss.fraunhofer.de

Torsten Blochwitz Thomas Neidhold
ITI GmbH
Webergasse 1, 01067 Dresden, Germany
torsten.blochwitz@iti.de thomas.neidhold@iti.de

Abstract

The paper presents the use of a subset of UML Statecharts to model discrete control components together with the physical model within a Modelica simulation environment. In addition, we show how statecharts can also be used to describe assertions charts for checking the compliance of user defined model properties and model behaviour during simulation. As the main difference to other approaches, neither Modelica language enhancements nor special libraries are necessary. The statechart model is automatically mapped onto standard Modelica constructs and can be simulated with any common Modelica standard simulator. Controlled by the user, the Modelica model can be automatically instrumented by additional Modelica code to examine the state coverage and transition coverage during simulation.

Keywords: state machine; statechart; control system, assertions, state coverage, transition coverage

1 Introduction

The modelling of discrete and hybrid control algorithms [1] is not a novel application area for Modelica. In the last years, Modelica libraries for Petri Nets [2] [4], Statecharts [3] or StateGraph [5] were introduced. Furthermore, the extension of Modelica with a new statechart section is discussed in [6].

In this paper, we present a new approach for modeling and verification of discrete control components within a Modelica environment. In contrast to the solutions mentioned above, we create the control component models of the physical system outside Modelica. The other modules of the physical system

are modeled as usual in the Modelica environment. In a second step, Modelica standard code is generated for the control components automatically. The insertion of the generated code into the Modelica physical model completes the system model (Fig. 1).

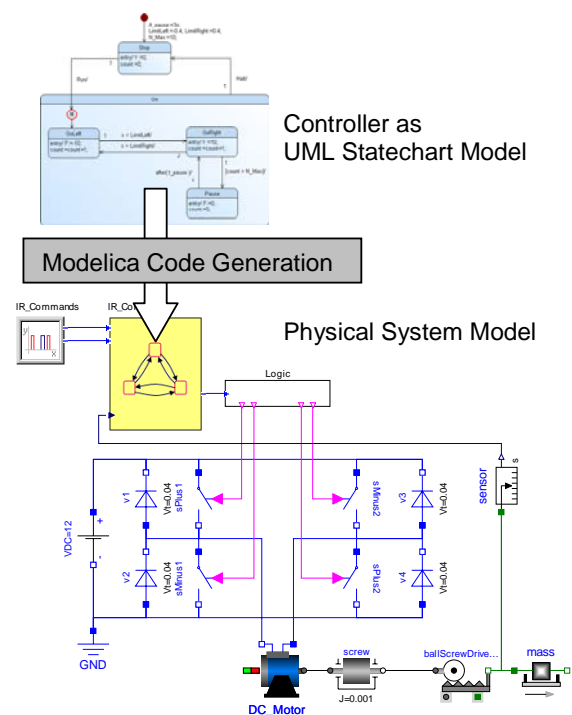


Fig. 1: Using UML Statecharts in SimulationX [9]

As modeling language for the control components we use a subset of UML Statecharts [7]. We derived the subset from an analysis of typical control algorithms in the domains mechanical and automotive engineering.

Besides control components, UML Statecharts proves to be suitable for robust modeling of physical

effects or technical sub-systems with discrete states (friction, hysteresis, valves, switches, etc.).

For our approach we see following advantages:

- UML Statecharts are well established for modeling of control algorithms, especially for reactive systems.
- The statechart creation outside Modelica allows the use of off-the-shelf UML development tools.
- The approach provides not only an interface to Modelica. The approach is also open to interface specialized verification tools esp. formal verification tools.
- In the sense of model based design, the approach is expandable for generation of production code for different targets such as PLCs or embedded controllers.
- The generated Modelica code can be simulated with any common Modelica simulator.

For UML Statechart entry, an additional Graphical User Interface (GUI) containing a UML Statechart editor is necessary which comes usually with the UML development tool. The UML tool should meet following requirements:

- The GUI as well as the UML tool code generator needs the ability to be customized.
- The UML tool should support the interaction between GUI and generated code to establish a comfortable visualization and animation.

In section 5 we present an UML Statechart editor which is completely integrated into the GUI of SimulationX [9].

The paper is organized as follows. Section 2 gives an overview on the supported UML Statechart subset. In section 3 some techniques are introduced which allow an efficient verification of the statechart models. Section 4 presents a prototypic implementation of our approach. An outlook on future work is given in section 5.

2 UML Statechart Subset

In this section we present the subset of UML Statecharts which is implemented in our prototype (see section 4). The subset contains the minimum of UML Statechart constructs to model a control component in a comfortable way:

- States: Simple States, Non-Concurrent Composite States, Pseudo States.
- Transitions: Signal Triggers, Change Triggers, Time Triggers, Guards.

- Activities: Modelica text.

The UML Statechart subset as well as the resulting Modelica code is illustrated with a linear drive as an example.

2.1 Example

The linear drive (Fig. 2) is controlled by the Controller module. Inputs for the Controller are the operator commands Run and Halt as well as the position x of the linear drive. As output, the controller delivers the DC motor supply voltage $U=-10V$ for left run, $U=+10V$ for right run and $U=0V$ for stop.

The specification of the controller is such as follows:

- Start after Run is given and drive to left
- Run 10 times between left and right end position
- Pause 3 seconds, afterwards continue
- Stop immediately after command Halt was given
- Restart with the action which was suspended after Halt, when Run is given again.

The physical system model of the linear drive example depicted in Fig. 2 is a simplification of the more complex model shown in Fig. 1.

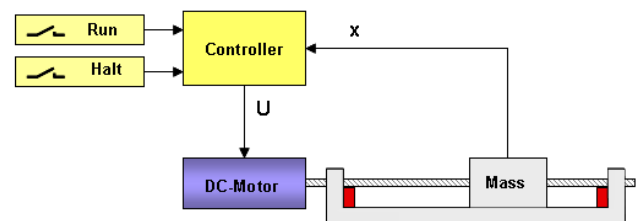


Fig. 2: Over-all structure of the linear drive

2.2 States

The control program (Fig. 3) is divided into the states Stop and Go. Stop is a simple state, whereas Go is a composite state with the nested simple states GoLeft, GoRight, and Pause. In consideration of hierarchy, the graph with the states Stop and Go is the top-level graph, implicit denoted as Main. The sub-jacent graph comprises the sub-states of Go.

For Modelica representation of state activities and state transitions, a state variable is declared for each hierarchy level. Their type declarations contain the enumerations of the state names. Each composite state is added with the enumeration InActive to indicate the inactivity of the composite state.

2.2.1 Simple States

In our subset, simple states may optionally have entry-activities, exit-activities, and activities which are initiated by internal transitions. These activities are simple Modelica algorithms. Modelica when-clauses are not allowed here.

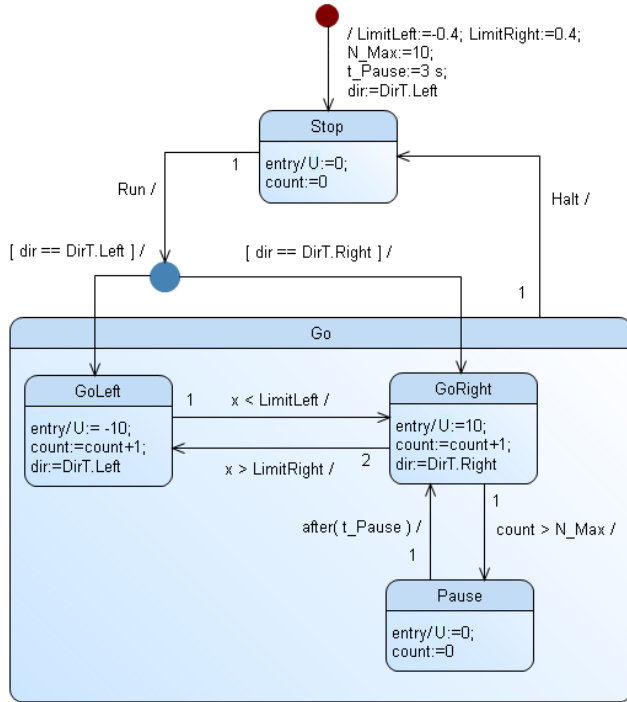


Fig. 3: Statechart model of the control program

2.2.2 Composite States

In comparison to simple states, composite states are extended each with a composition compartment. In our approach, this compartment comprises only one region of sub-states – this means, concurrency of activities can not occur within one statechart instance. Concurrency is only possible between multiple statechart instances.

In the generated Modelica code, all entry-activities of the nested sub-states are gathered in a when-clause separately (see 2.4 Entry-activities of Go). All other activities are included in an if-clause which describes the transitions of the composite state.

2.2.3 Pseudo States

We support following pseudo states: initial state, junction, and shallow history. An initial state indicates the default starting point of processing the statechart or a composite state. A junction merges multiple incoming transitions into a single outgoing transition, or conversely, split an incoming transition

into multiple outgoing transitions. A shallow history stores the most recent active sub-state of a composite state after leaving it. When the composite state is newly entered via shallow history this sub-state becomes active again.

2.3 Transitions

Following kinds of transitions may be used: simple transitions (connecting two states), self-transitions (the same state acts as both the source and the destination), compound transitions (connecting many states via junction pseudo states), group transitions (originating from composite states), and internal transitions of simple states. A trigger, a guard, and a transition-activity may label a transition.

The triggering of a group transition implies the exiting of all the sub-states of the composite state and executing their exit-activities starting with the innermost states. An internal transition executes without exiting or re-entering the state in which it is defined.

In our approach, each transition is triggered with a single trigger as described below.

2.3.1 Signal Trigger

Generally, a signal trigger represents the receipt of an asynchronous signal instance [7]. In our interpretation, a signal is either a record typed message with e.g. one integer and real component or a boolean typed variable, typically a controller input command. Every new signal is notified by toggling a flag which is an additional component of the message. In case of a boolean variable, Run and Halt in the example, the variable itself is toggled.

Signals are produced either by modules of the physical system or inside the statechart instance.

Signal type definition:

```

type SignalT = record SIGNAL
  Boolean flag;
  Integer int_val;
  Real real_val;
end SIGNAL;
    
```

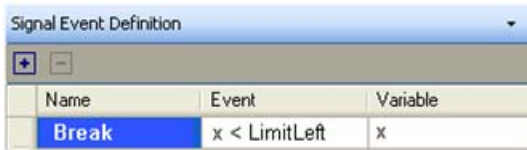
Signal assignment in physical system module:

```

SignalT Run;
when ( time >= 1 ) then
  Run.flag := not Run.flag;
  Run.int_val := 100;
end when;
    
```

A toggled signal is detected by the Modelica change-function, for instance *change (Run.flag)*.

In addition to UML, signals can be also defined in a Signal Definition Table (Fig. 4). In the table, signal events are derived from the achievements of predefined thresholds of physical system quantities or of internal statechart variables.



Name	Event	Variable
Break	$x < \text{LimitLeft}$	x

Fig. 4: Example of the Signal Definition Table

2.3.2 Change Trigger

A change trigger specifies an event that occurs when a boolean-valued expression becomes true as a result of a change in value of one or more attributes [7]. In UML the life time of the change event is a semantic variation point. Related to control tasks, in our approach the change event remains true as long as the evaluation of the change expression results in true. In our linear drive example, change triggers are $x < \text{LimitLeft}$, $x > \text{LimitRight}$, and $\text{count} > N_Max$.

The Modelica representation of this behavior is given by an if-clause in the transition block (see 2.4 Transitions of Go).

2.3.3 Time Trigger

A time trigger specifies a time event, which models the expiration of a specific deadline [7]. We restrict the deadline to a relative expression. The expression is relative to the time of entry into the source state of the transition triggered by the event, e.g. after (t_Pause). The time event is generated only if the state machine is still in that state when the deadline expires.

In Modelica this behavior is reflected in following steps: Firstly, if the source state entry is detected in a when-clause, a time variable is set to the time limit (see 2.4 Entry-activities of Go). Secondly, a when-clause checks if the simulation time exceeds the time limit. If true, a timeout signal is toggled. This when-clause belongs to the event generation block of the module (see 2.4 Event generations). Thirdly, if the source state will inactive due to another transition, the time variable is reset.

2.3.4 Guards

A guard is a Boolean expression written in terms of parameters of the triggering event or attributes of the context object [7]. It is evaluated only once whenever it's associated event fires. If it is false, then the

transition does not fire and the event is lost. In the linear drive example, the guards $[\text{dir} == \text{DirT.Left}]$, $[\text{dir} == \text{DirT.Right}]$ determine the target sub-states of Go after the Run command is given.

2.3.5 Firing Priorities

It is possible that more than one transition could be concurrently fired to change the state, e.g., they have the same trigger event and their guard expressions results in true. Then, in UML, an implicit priority rule is applied based on the relative position of the source state in the state hierarchy [7]. In addition, we allow the user to assign the transition priorities explicitly. A transition priority is denoted by a number 1, 2, 3... where 1 symbolizes the highest priority. These priorities are depicted near the start points of the transition arrow lines. Chosen by the user, the priorities of group transitions are either higher or lower than priorities of inner transitions of composite states.

The resulting priority number determines the position of the transition in the check for firing. In the linear drive example, Halt shall have the highest priority to stop the machine, especially in case of an emergency.

2.4 Over-all Modelica Code Structure

The Modelica representation of a statechart consists of following sections:

- Declaration of state variables, input/output signals, internal signals, system variables, auxiliary variables, parameters.
- Initialization of state variables and auxiliary variables, execution of initial transition activities (when-clause).
- Event generation block: generation of signal events according to signal event definition table, generation of timeout events (when-clauses).
- Entry-activity block: detection of state entries, execution of entry-activities, assignment of time limits to time variables, generation of completion events of composite states (when-clauses).
- Transition block: event detection, assignment of next state, execution of exit-activities and transition-activities, reset of time variables (if-clauses).

For the linear drive example the Modelica code is given below. To shorten, the declaration section is omitted.

Module Controller

Initialization:

```
when initial()then
  mainState:=MainStateT.Stop;
  ontoMainState:=MainStateT.Stop;
  goState:=GoStateT.InActive;
  entryGoState:=GoStateT.InActive;
  LimitLeft:=-0.4; LimitRight:=0.4;
  t_Pause:=3;
  t_PauseFinished:=0;
  count:=0; N_Max:=10;
  dir:=DirT.Left;
  timeout:=false;
  completeGo:=false;
end when;
```

Event generations:

```
when (time>=t_PauseFinished) then
  timeout:=not timeout;
end when;
```

Entry-activities of Main:

```
when (mainState==MainStateT.Stop) then
  U:=0; count:=0;
elsewhen (mainState==MainStateT.Go) then
  goState:=entryGoState;
end when;
```

Entry-activities of Go:

```
when (goState==GoStateT.GoLeft) then
  U:=-10; count:=count+1;
  dir:=DirT.Left;
elsewhen (goState==GoStateT.GoRight) then
  U:=10; count:=count+1;
  dir:=DirT.Right;
elsewhen (goState==GoStateT.Pause) then
  U:=0; count:=0;
  t_PauseFinished:=time+t_Pause;
elsewhen (goState==GoStateT.InActive) then
  completeGo:=not completeGo;
end when;
```

Transitions of Main:

```
if (pre(mainState)==MainStateT.Stop) then
  if (change(Run)) then
    if (dir==DirT.Left) then
      mainState:= MainStateT.Go;
      entryGoState:=GoStateT.GoLeft;
    elseif (dir==DirT.Right) then
      mainState:= MainStateT.Go;
      entryGoState:=GoStateT.GoRight;
    end if;
  end if;
elseif(pre(mainState)==MainStateT.Go)then
  if (change(completeGo)) then
    mainState:=ontoMainState;
  end if;
end if;
```

Transitions of Go:

```
if (pre(goState)==GoStateT.GoLeft) then
  if (change(Halt)) then
    goState:=GoStateT.InActive;
    ontoMainState:=MainStateT.Stop;
  elseif (x<LimitLeft) then
    goState:=GoStateT.GoRight;
  end if;
elseif(pre(goState)==GoStateT.GoRight)then
  if (change(Halt)) then
    goState:=GoStateT.InActive;
    ontoMainState:=MainStateT.Stop;
  elseif (count>N_Max) then
    goState:=GoStateT.Pause;
  elseif (x>LimitRight) then
    goState:=GoStateT.GoLeft;
  end if;
elseif (pre(goState)==GoStateT.Pause)then
  if (change(Halt)) then
    t_PauseFinished:=time;
    goState:=GoStateT.InActive;
    ontoMainState:=MainStateT.Stop;
  elseif (change(timeout)) then
    goState:=GoStateT.GoRight;
  end if;
end if;
```

```
end Controller;
```

3 Verification

The main tool for the verification of the Modelica model is the simulator. In this section we describe techniques to increase the efficiency of the simulation based verification: Design Rule Check, State Coverage Analysis and Transition Coverage Analysis, and Assertion Charts.

3.1 Design Rule Check

During graphical entry and compilation of statecharts the following design rules are currently checked:

- Only one initial state is allowed on each hierarchy level.
- An initial state has exactly one outgoing transition. Trigger and guards are not allowed.
- Pseudo states must not be connected by transitions.
- A split junction has only one incoming transition. Only this transition has a trigger.
- A merge junction has only one outgoing transition. Only this transition has a trigger.
- Self-transitions are not allowed for composite states.
- Each state, except initial state, has at least one incoming transition.

- Isolated sub-graphs are not allowed.
- A warning is given when some outgoing transitions of one state have the same priority number.

3.2 State and Transition Coverage

The analysis of both state activations and transition activations are measures for the achieved functional coverage during the simulation run.

To measure the activation of states and transitions, additional Modelica code is automatically inserted into generated Modelica code (described in section 2.4) by the Modelica code generator (see section 4.2). The additional code basically consists of a set of counters. A unique counter is associated with any entry-activity block and any transition block. The counters are implemented as integer vectors. Every state or transition activation leads to an increment of the associated counter component.

For comfortable use, the actual achieved counting results may continuously back-annotated into the UML Statechart schematic.

3.3 Assertion Charts

Our Assertion Chart approach is derived from [10]. Assertion Charts aim for watching the behavior of the system in respect of specified properties. With the help of Assertion Charts, we describe both non-temporal and temporal system properties which have to be examined.

For the linear drive, e.g., an assertion may be: If the DC motor supply voltage U alters to $U > 0$, then drive position x will exceed its limit within T seconds, otherwise a fault will be reported (Fig. 5).

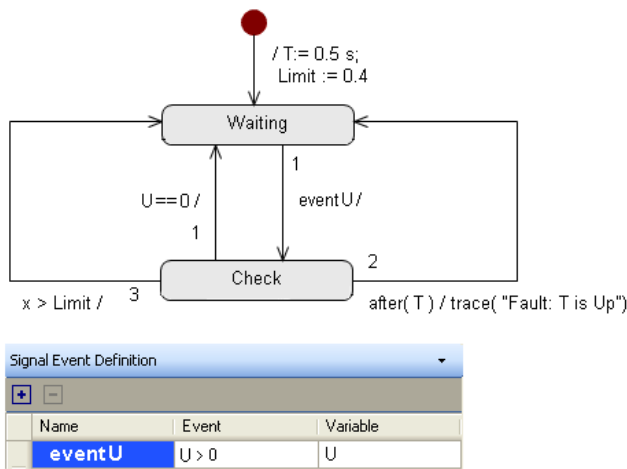


Fig. 5: Assertion Chart example of the linear drive

In our case, the charts are composed like ordinary statecharts and run simultaneously to them. We deliver a set of typical Assertion Charts concerning dedicated event sequences and time limits. The predefined Assertion Charts are parameterized.

4 Implementation Prototype

The section introduces the first implementation of our approach into the SimulationX Modelica environment [8][9]. The implementation consists of the components Statechart Editor, Modelica Code Generator, Run Time Visualization (simulation driven statechart animation, association of the State Coverage and Transition Coverage Analysis results). All additional functionality can be controlled by the user within the SimulationX GUI. For easy use, a set of typical control domain Assertion Charts comes with the implementation.

4.1 Statechart Editor

The Statechart Editor (Fig. 6) is seamlessly integrated into the SimulationX framework. The Statechart Editor has the following features:

- Schematic entry of the statecharts,
- Definition of statechart activities as Modelica code,
- Definition of triggers,
- Definition of local types, variables and parameters,
- Definition of the interface to the physical system.

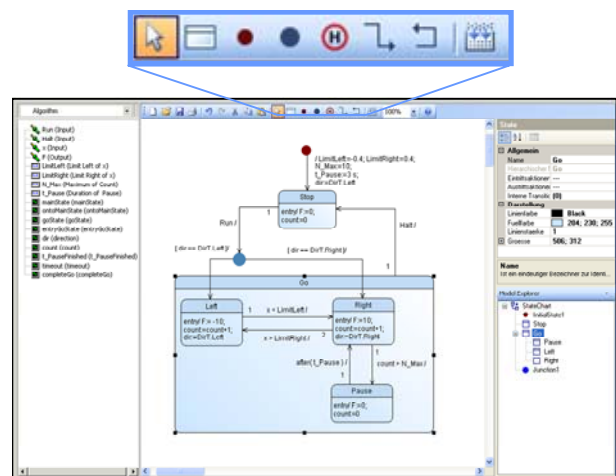


Fig. 6: Statechart Editor window

Pressing the compile button starts the Modelica Code Generator. The structure of code is outlined in section 2.4.

4.2 Modelica Code Generator

The Modelica Code Generator generates standard Modelica code from the UML Statechart model. The code is user-driven instrumented with additional code to perform the State Coverage Analysis and Transition Coverage Analysis and to associate Assertion Charts with selected signals. In addition, the Modelica Code Generator accomplishes the Design Rule Check. Besides Modelica code the generator architecture is open to generate code for various targets such as formal verification tools or production code for PLCs or embedded controllers.

4.3 Runtime Visualization

SimulationX supports the visualization of statecharts at runtime by a specialized view. During simulation the active state and the latest transition are highlighted if a time step or time interval is completed (Fig. 7). Controlled by the user, state and transition coverage are also displayed. The actual state and transition counter readings are annotated on the states and on the end points of transitions, respectively.

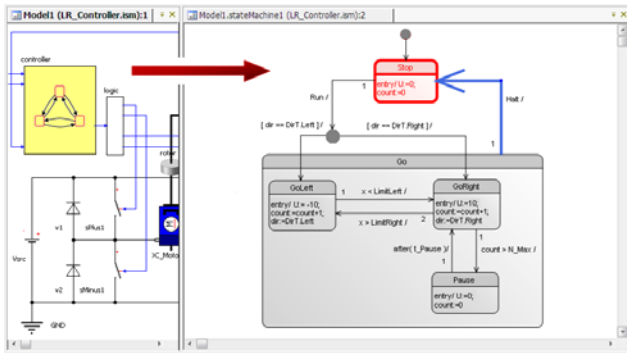


Fig. 7: Back-annotation of active states

For checking the functionality of whole system, the common view of internal controller variables and variables of the equipment are essential. Therefore all variables of the statechart, inclusive the state variables, may be accessed by the user.

In case of the linear drive example the causality of the system behavior (Fig. 8) is reflected by the signals Run and Halt of the operator, the variables goState and count of the controller, its output U, and the position x of the drive. The correlations between these quantities may be checked by assertion charts during the simulation.

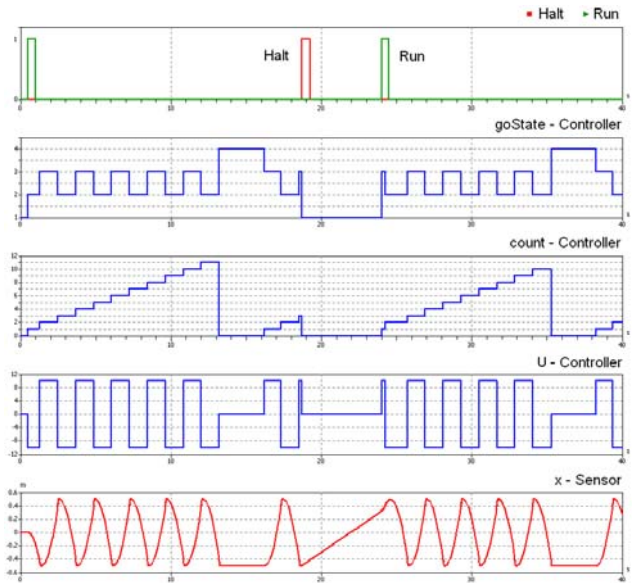


Fig. 8: Wave-forms of the linear drive example

5 Conclusions

We have applied our methodology to control systems in the area of automotive, robotics and manufacturing systems engineering. The approach introduced in the paper proves to be very comfortable for modeling of control components and physical system within the SimulationX environment. Especially the integrated verification support decreases modeling and simulation cycles and leads to robust control components with high test coverage in relative short time. Our next steps are the successive extension of the approach to a Modelica model based design environment for control algorithms.

Next steps are:

- Enhancements in code generation

Right now, we generate Modelica code to validate the control component behavior within the physical system. For the implementation of the control component models, target code has to be derived from the models according to the target hardware platform, e.g. for PLC systems or embedded systems.
- Enhancements in formal verification

In the recent years, considerable progress has been achieved in the field of formal verification tools for model checking especially for digital integrated circuit verification. Therefore, we are encouraged to interface our modeling approach to selected model checker tools by transformation of our statechart models to the model checker input description.

References

- [1] Donath, U.; Haufe, J.; Schwarz, P.: Mehr-Ebenen-Simulation automatisierungstechnischer Prozesse und Steuerungen (in German). 4. Fachtagung Entwurf komplexer Automatisierungssysteme, Braunschweig, pp. 543-553, June 7-9, 1995
- [2] Mosterman, P.J., Otter, M., Elmqvist, H.: Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. Summer Computer Simulation Conference '98, Reno, Nevada, USA, pp. 314-319, July 19-22, 1998
- [3] Ferreira, J.A., de Oliveira, J.E.: Modeling Hybrid Systems Using Statecharts and Modelica. 7th IEEE International Conference on Emerging Technologies and Factory Automation, Barcelona, Spain, 18-21 Oct., 1999.
- [4] Fabricius, St.: Extensions to the Petri Net Library in Modelica. <http://www.modelica.org>, 2001
- [5] Otter, M.; Årzén, K.-E.; Dressler, I.: StateGraph - A Modelica Library for Hierarchical State Machines. Proc. 4th Int. Modelica Conf., Hamburg, Germany, March 7-8, 2005, pp. 569-578.
- [6] Nytsch-Geusen, Ch: The use of the UML within the modeling process of Modelica-models. Proc. ECOOP, Berlin, Germany, July 30, 2007.
- [7] UML Superstructure Specification, v2.0. <http://www.omg.org>
- [8] Neidhold, Th. et al: Modeling of State Machines in SimulationX. 10th ITI Simulation Workshop, Dresden, Germany, September 20-21, 2007.
- [9] <http://www.simulationx.com>
- [10] Drusinsky, D: Modeling and Verification Using UML Statecharts. Elsevier Inc., Oxford, 2006

¹ The project was founded by the European Regional Development Fund (ERDF) and by the Free State of Saxony in the framework of technology promotion.

Session 3b

Thermodynamic Systems & Applications

Model-based online applications in the ABB Dynamic Optimization framework

Rüdiger Franke
ABB Power Technology Systems
Mannheim, Germany

Marc Antoine
ABB Power Technology Systems
Baden, Switzerland

B.S. Babji
ABB Corporate Research
Bangalore, India

Alf Isaksson
ABB Corporate Research
Västerås, Sweden

Keywords: technical calculations, model-based simulation, model tuning, online optimization, model-based control, 800xA, Modelica, HQP

Abstract

ABB Dynamic Optimization is an extension for the control system 800xA. Exploiting the Aspect Objects technology of 800xA, Dynamic Optimization allows the seamless integration of model based applications, such as model-based process control. Running offline and online using one and the same software environment, Dynamic Optimization provides an attractive framework to apply offline results online, in order to optimize the efficiency of an industrial process.

1 Introduction

Industrial plants search for powerful diagnostic and optimization tools to monitor and predict plant performance, issue early warnings for equipment diagnosis, sensor validation and preventive maintenance. New modeling technologies and increasing computational power make the online application of computer simulations more and more attractive.

This paper shows how the Modelica technology is exploited in the Industrial IT Extended Automation System 800xA using the Dynamic Optimization framework.

2 Dynamic Optimization framework

2.1 Industrial IT System 800xA

The architectural framework for the Industrial IT System 800xA is built upon ABB's Aspect Object tech-

nology. Aspect Objects relate plant data and functions – the aspects, to specific plant assets – the objects. Aspect objects represent real objects, such as process units, devices and controllers. Aspects are informational items, such as I/O definitions, engineering drawings, process graphics, reports and trends that are assigned to the objects in the system.

Aspect Objects are organized in hierarchical structures that represent different views of the plant, such as Functional Structure and Location Structure. One object may be placed multiple times in different structures. The idea of placing the same object in multiple structures is based on the IEC standard 1346 [7, 2].

2.2 Dynamic Optimization architecture

Figure 1 gives an overview about the software architecture. Many existing software interfaces and components that are intended for process control are re-used by Dynamic Optimization for model-based applications. These are in particular the connectivity to a process and the treatment of trend&history data using OPC DA and OPC HDA technology, respectively. Furthermore, these are operator graphics, alarms and events, as well as a common configuration database.

Dynamic Optimization adds new software components for the management of model knowledge and for running model-based activities. Model variables are treated like process signals by allocating OPC properties in a simulation server.

The models are being built and tested with a modeling application such as Dymola or MathModelica. Afterwards executable model code is exported to Dynamic Optimization. The numerical solver HQP is employed at runtime. HQP combines a large-scale nonlinear optimization solver with ODE and DAE solvers, see e.g. [5].

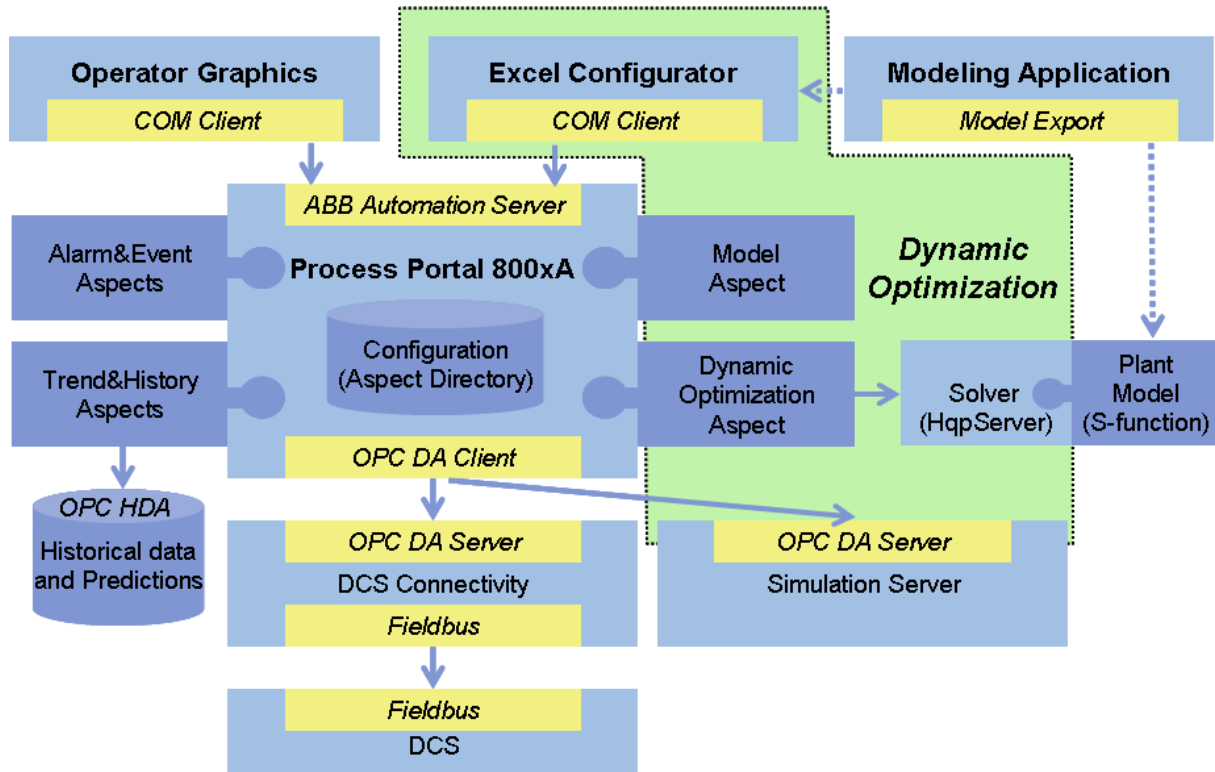


Figure 1: Software architecture of the Dynamic Optimization framework.

Excel serves as intermediate data layer between the modeling application and the control system. This allows to make small changes, e.g. to change parameter values, at runtime. Moreover advanced model based activities, including optimization objectives and constraints not covered by Modelica, can be specified in Excel. Furthermore, an activity can be archived in an Excel file, including the parameterization of the model, solver settings, used process data, and results.

The use of standard interfaces and independent applications for modeling and numerical solution does not only provide for seamless integration re-using existing tools, but it also gives more flexibility regarding the framework itself. This way for instance research studies can be done in a platform more appropriate for research. As the same tools are used, the platform can easily be changed to deploy research results later on.

Take the industrial case study [9] for example. The research on model predictive control of batch processes has been done using Matlab and Simulink as platform, together with the HQP numerical solver and OPC process interfaces [9]. Using Dynamic Optimization, the research results can be deployed in the control system 800xA by running the same model code and using the same solver.

2.3 Introductory example

Consider one wants to perform a simple calculation online, like the determination of the efficiency of a preheater in a power plant. According to water-tube boilers standards [4], the efficiency of a heat exchanger is defined as

$$\eta = \frac{T_{22} - T_{21}}{T_{11} - T_{21}} \quad (1)$$

with T_{11} and T_{21} the inlet temperatures at primary and secondary side, respectively, and T_{22} the outlet temperature at the secondary side.

The heat exchanged in a preheater mostly results from the condensation of the fluid at the primary side. This is why the inlet temperature T_{11} is not used directly from a measurement. It is replaced by the saturation temperature $T_{\text{sat}}(p_{11})$ for a measured pressure p_{11} at the primary side. This gives a more reliable value for the preheater efficiency.

The calculation can be set up graphically in a Modelica tool, see Figure 2. The block `heatExchangerEfficiency` contains the general calculation given in (1). The saturation temperature $T_{\text{sat}}(p_{11})$ is calculated for a measured p_{11} using a function available in the standard Modelica.Media library.

In order to run the calculation in the control system 800xA, first appropriate Aspect Objects have to set

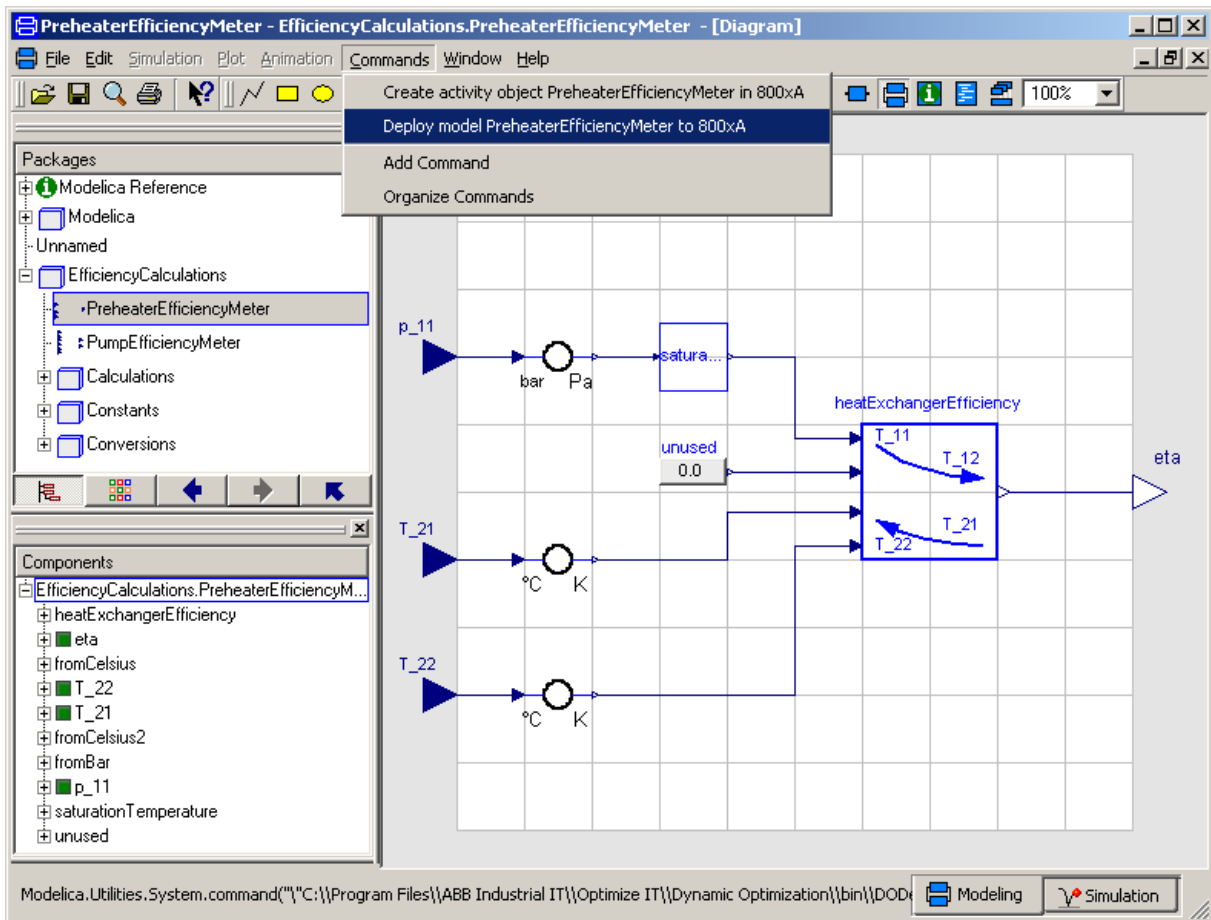


Figure 2: Graphical implementation of a preheater efficiency calculation in Dymola.

The screenshot shows the 'Plant Explorer Workplace' for the 'PreheaterEfficiencyMeter' model. The 'Aspects of PreheaterEfficiencyMeter' table is displayed below.

	Category	Name	Value	Unit	Ref'd Object	Ref'd Aspect	Ref'd Proper	Description
1	parameter	heatExchange	1					Relate efficiency to
2	parameter	heatExchange	-1					Gain of upper input
3	parameter	heatExchange	1					Gain of lower input
4	parameter	heatExchange	1					Gain of upper input
5	parameter	heatExchange	-1					Gain of lower input
6	parameter	heatExchange	1					Gain of upper input
7	parameter	heatExchange	-1					Gain of lower input
8	output	eta	0	1				efficiency of prehe
9	input	T_22	140	degC				Outlet temperature
10	input	T_21	110	degC				Inlet temperature at
11	input	p_11	5	bar				Pressure at primary

Below the table, the 'All Variables' section is visible, and the bottom status bar shows '800xAService' and the ABB logo.

Figure 3: PreheaterEfficiencyMeter in 800xA Plant Explorer Workplace.

up. They can be created from the modeling application Dymola by invoking the menu command “Create aspect object PreheaterEfficiencyMeter in 800xA”. The menu command is included as annotation in the model, being accessible from Dymola and calling the according system command provided by Dynamic Optimization. The hierarchical structure of Aspect Objects in 800xA corresponds to the package structure in Modelica, see Figures 2 and 3.

The model can be deployed with the menu command “Deploy model PreheaterEfficiencyMeter to 800xA”. The deployment process consists of multiple steps that have been automated. First the model is translated and an executable dynamic link library is created. Three aspects are configured in 800xA for the Aspect Object PreheaterEfficiencyMeter, see Figure 3 and the relation to the software architecture shown in Figure 1:

1. The Modelica Model aspect declares the interface of the model through inputs, outputs, parameters and states. Moreover the model aspect contains attributes like value, unit and description for each model variable. The empty columns “Ref’d Object”, “Ref’d Aspect”, “Ref’d Property” can be used to link model variables to signals in the control system.
2. The Control Connection aspect declares OPC properties for the model variables, which are served by the Simulation Server.
3. The Dynamic Calculation aspect links the model to the solver and defines solver settings like sampling rate for the calculation.

Besides the aspects, also the Simulation Server for OPC properties introduced by the model and the Dynamic Optimization Service running HQP solver instances are updated during deployment.

3 Model-based applications

The use of Modelica gives access to a broad range of possible applications using a common modeling technology.

3.1 Technical calculations

Many calculations that are of interest for industrial applications have been standardized. The water-tube boilers standards are an example [4]. The standardized formulae and algorithms can be implemented in Modelica and evaluated online. This provides e.g. for the

online determination of thermal stress and the implementation of lifetime counters for critical components of a power plant. Further applications cover the online determination of the plant efficiency.

Such technical calculations are normally set up to run automatically on a specified sample time interval. The results of the calculations are added as new signals to the control system.

3.2 Model-based simulation

Modelica is designed to allow convenient, component-oriented modeling of complex physical systems. Available model libraries significantly simplify the object-oriented modeling process. Such models can run online to analyze and optimize the modeled process. Further applications of model-based simulation include the training of plant operators and the validation of automatic controllers.

Take soot blowing, a cleaning mechanism for steam boilers, as example. Due to constant fouling, the heat transfer coefficient in superheaters and the steam pressure decrease during the operation of a recovery boiler. Soot blowing is applied to clean the boiler.

The modeling and simulation study [8] describes the dynamic modeling of a recovery boiler of a paper mill in Modelica, in order to simulate the effects of fouling and soot blowing on drum pressure and heat transfer coefficients.

3.3 Estimation and optimization

Prior to an online application, a model normally needs to be tuned. Unknown model parameters, such as heat transfer coefficients, need to be estimated based on process data resulting from experiments. This can be done using a Dynamic Estimation aspect. The experiment data is directly available in 800xA through history Log Configuration aspects. The estimation setup can be archived together with the used process data, solver settings and estimation results in an Excel file. Moreover estimation can be configured to run online, in order to analyze the process or to validate measurements. Future development may involve addition of model validation against separate data sets as well as adding grey-box calibration methods similar to the ones described in [1].

Having a well tuned model, it can be used to optimize the modeled process, such as the optimization of steady-state setpoints or of transient control trajectories. The Dynamic Optimization aspect allows to setup and solve optimization problems for a given model.

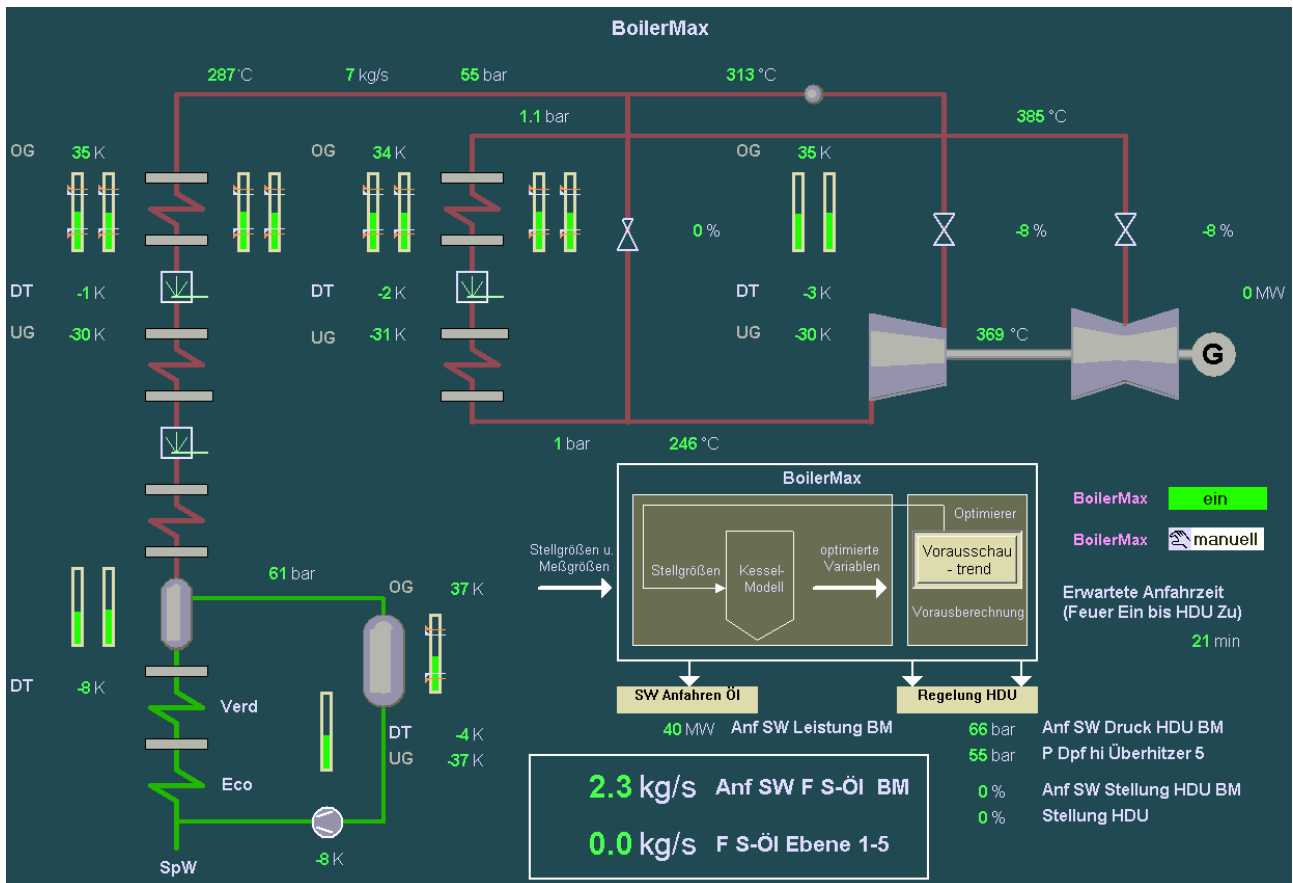


Figure 4: Operator display for boiler startup optimization.

The solver HQP covers initial-value simulation problems for hybrid DAEs resulting from the translation of a Modelica model. Mathematical optimization and estimation problems, however, can currently only be solved for a simplified hybrid DAE, containing no state events. A formulation and solution of mixed-integer nonlinear optimization problems, as required for instance for planning&scheduling of production processes, may be subject of future development.

3.4 Model-based control

Applying Model Predictive Control (MPC) technology, a model can be used to predict the optimal operation of the process. The calculated predictions get applied in closed control loop. An example application, that has been installed successfully in several power plants, is the startup optimization of large steam boilers [6, 5]. Appropriate process models can be built based on the Modelica.Media and Modelica.Fluid libraries [3].

Figure 4 shows an operator display used for boiler startup optimization. Elements known from a regular

operator graphics, such as temperature measurements and operator controls, are seamlessly integrated with new elements resulting from the model based BoilerMax application, such as thermal stress values, stress limits and predicted startup time. Furthermore, the results of the predictive startup optimization are accessible in a regular trend display showing the optimization results as future process data.

3.5 Runtime scheduling and supervision

Individual model based activities can be performed for a specified sample time or triggered by changing process data on-line. However, more advanced runtime scheduling might be required for complex activities. For instance, one might want to synchronize multiple activities, supervise solver times and implement specific error recovery strategies. Such advanced runtime scheduling can be formulated in Modelica as state graph [10]. In the Dynamic Optimization framework, a runtime scheduler is running as additional activity, triggering and supervising other activities, such as state estimation and predictive optimization.

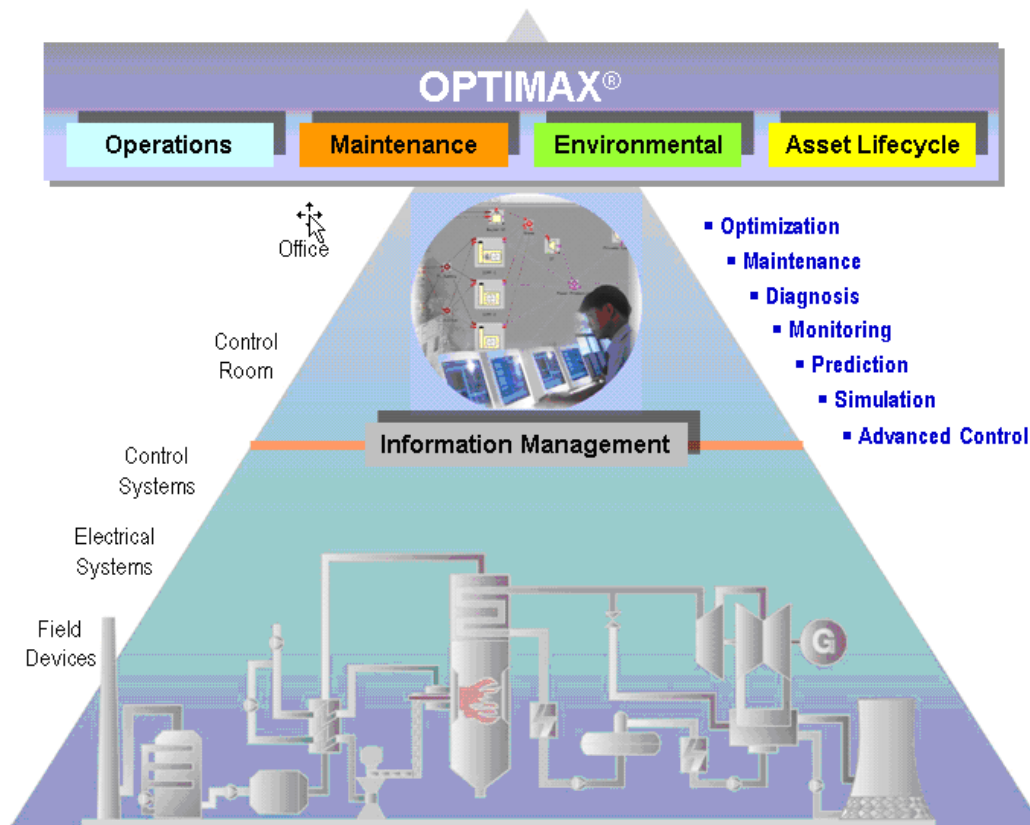


Figure 5: Overview about the ABB solution OPTIMAX®.

4 Optimizing energy efficiency in the power industry

The primary key to energy efficiency in the power industry is reducing the cost of fuel and consumables. Industrial plants are huge energy consumers; therefore small percentage savings can have a significant impact on their bottom line. Figure 5 gives an overview about the ABB solution OPTIMAX®, which uses Dynamic Optimization as one tool, besides others.

4.1 Maximize operational performance & efficiency

OPTIMAX® operations solutions can handle utilities with complex generation portfolios which are seeking to minimize energy generation costs, be it electrical or a combination of electrical and other forms of energy. In addition, deciding whether or not it makes sense to buy or sell power or fuel, start or stop a unit, save lifetime, or postpone a preventive maintenance outage can be easily answered.

4.2 Minimize maintenance cost

Maintenance expenses are second only to fuel costs as variable costs. The key to optimizing assets is often having information that is accurate, timely and actionable. Clearly, the ability to act on reliable information is as essential as having access to the information in the first place.

Work preparation and planned condition-based maintenance are increasingly important for reduction of downtime. The benefit of OPTIMAX® Maintenance Management Solutions is to achieve and maintain a high level of availability, quality and safety of the plant. This applies to current plant operation but is particularly valid for inspection, overhaul and service activities. For industrial users this leads to a higher Return On Asset (ROA) which is a key driver of shareholder value.

4.3 Reduce emissions & waste

The measurement and reduction of hazardous emissions is increasing in importance and regulatory standards are getting stricter every day. Emission of green-

house gases now has measurable economic value and operators have a real incentive to lower these emissions. The OPTIMAX[®] environmental solutions reduce emissions by monitoring flame quality, measuring coal flow and carbon in ash content, and providing Advanced Process Control (APC) which optimizes combustion, shortens boiler startup times and improves efficiency.

4.4 Extend the Asset Life Cycle

From an economic perspective, plant managers seek to balance their investment in new assets against performance, risk and downtime. OPTIMAX[®] solutions for lifecycle optimization of assets are able to schedule the most economical operation of different generating units and trade-off income from sales against lifecycle costs. In addition, this approach is also capable of taking emission costs into account, i.e. more stringent CO₂ requirements may make plants that are still mechanically functional uneconomic to run. The advantage of these decision support tools is the ability to include plant ageing models to find the optimal operational strategy between maintenance outages, especially when operating under environmental constraints.

5 Conclusions

The Dynamic Optimization framework allows running model-based applications in the Industrial IT Extended Automation System 800xA. The Modelica technology and the Aspect Object technology of System 800xA are integrated seamlessly. Dynamic Optimization provides software components that establish links between industrial control and the world of physical modeling and simulation.

Simple model based simulations can be deployed directly from a graphical modeling application, such as Dymola or MathModelica. Excel is used as intermediate data layer. This allows the treatment of advanced solver settings, such as optimization constraints, and of process data used for offline applications, such as experiment data used for parameter estimation.

A broad range of model-based applications is becoming possible. Examples are the simulation of recovery boilers in pulp mills [8], model predictive control of batch processes [9], and cost optimal startup of power plants [5].

The optimization of energy efficiency is a particular application area of high interest.

References

- [1] T. Bohlin. *Practical Grey-box Process Identification – Theory and Applications*. Springer, 2006.
- [2] L.G. Bratthall, R. van der Geest, H. Hoffmann, E. Jellum, Z. Korendo, R. Martinez, M. Orkisz, C. Zeidler, and J. S Andersson. Integrating hundred's of products through one architecture – the Industrial IT architecture. In *International Convergence on Software Engineering*. Orlando, Florida, USA, 2002.
- [3] H. Elmqvist, H. Tummescheit, and M. Otter. Modeling of thermo-fluid systems – Modelica.Media and Modelica.Fluid. In *Proceedings of the 3rd International Modelica Conference*. Modelica Association, Linköping, Sweden, November 2003.
- [4] European committee for standardization. Water-tube boilers standards. EN 12952, 2001.
- [5] R. Franke and L. Vogelbacher. Nonlinear model predictive control for cost optimal startup of steam power plants. *at – Automatisierungstechnik*, 54(12):630–637, 2006.
- [6] R. Franke and B. Weidmann. Steaming ahead with optimizing power plant boiler startup. *Power Engineering International – PEI*, 15(6), 2007.
- [7] International Electrotechnical Commission. Industrial systems, installations and equipment and industrial products – structuring principles and reference designations. IEC Standard 61346, 1996.
- [8] Tarun Prakash Mathur and B.S. Babji. Object oriented approach to dynamic modeling of recovery boiler in Modelica. In *Proceedings of Intl. Conf. on Modeling and Simulation*. Coimbatore, India, August 2007.
- [9] Z.K. Nagy, B. Mahn, R. Franke, and F. Allgöwer. Nonlinear model predictive control of batch processes: an industrial case study. In *16th IFAC World Congress*. Prague, Czech Republic, July 2005.
- [10] M. Otter, J. Årzén, and A. Schneider. State-Graph – a Modelica library for hierarchical state machines. In *Proceedings of the 4th International Modelica Conference*. Modelica Association, Hamburg-Harburg, Germany, March 2005.

Using Modelica/Matlab for parameter estimation in a bioethanol fermentation model

Juan I. Videla Bernt Lie

Telemark University College

Department of Electrical Engineering, Information Technology, and Cybernetics

Porsgrunn, 3901 Norway

Abstract

Bioethanol production from fermentation of a substrate using biomass as catalyst is considered. Four alternative reaction rate models with different levels of details are derived and implemented in Modelica. The problem of parameter estimation of models using state/parameter estimation techniques in a Modelica-Dymola/Matlab setup is discussed. Practical aspects concerning the different implementations of nonlinear estimators are analyzed (EKF, UKF, and EnKF). The use of Modelica-Dymola for “on-line” applications such as state estimation poses the additional problem of the efficiency of the code; this will also be discussed. The four reaction rate models are fitted using fictitious experimental data generated from one of the models to illustrate the parameter estimation procedure.

Keywords: bioethanol fermentation, parameter estimation, nonlinear estimators

1 Introduction

Alcoholic fermentation is an important biochemical process which has been known for some 5000 years. Ethyl alcohol, or more commonly ethanol, has chemical formulae C_2H_5OH , and finds uses as (i) alcoholic beverage (beer, wine, spirits), (ii) solvent, (iii) raw material in chemical synthesis, and (iv) fuel.

With the current focus on CO_2 release and global warming, there is a considerable interest in producing fuel from biomass. Production of ethanol from fermentation typically involves a two step process: (a) the main process where substrate (glucose) is converted to ethanol and non-fossil CO_2 in an enzymatic process, and (b) the aéro-

bic yeast growth through the consumption of substrate and oxygen.

In continuous reactors, yeast is continuously washed out, leading to a less efficient use of the yeast. The use of immobilized yeast increases the efficiency of the process, as less substrate is “wasted” for yeast production. In fermentation, salts are involved as co-enzymes. The resulting ions affect the oxygen uptake in the reaction mixture.

The produced (bio-) ethanol can be used as fuel after some additional processing –filter yeast, remove water by distillation, etc. Alternatively, the ethanol can be converted to methane by microorganism.

The efficient production of ethanol in a fermentation reactor requires quantitative analysis of how raw materials are converted to products. Static models are often used for design purposes for continuous reactors, while dynamic models are required for batch reactors (e.g. beer production) and for control analysis and design in continuous reactors. A simple numeric dynamic model for the continuous fermentation of glucose using the yeast *saccharomyces cerevisiae* is given in [1]. The model is somewhat simplified in that the dynamics of the overall reactor volume is neglected, the role of the salts as co-enzymes is neglected, and somewhat simple kinetic reaction rates are used. A more systematic development of reaction rates for the continuous ethanol fermentation process is presented in [2]. In [1], the effect of ions on the oxygen uptake in the reactor mixture is included, but the effect of glucose is neglected; expressions for the effect of ions and sugars are given in [3]. Most of the parameters of the model of [1] are given in their publication; however there is one or two typos, and the effect of salt ions on the

oxygen uptake is as if the salinity of the reaction mixture was similar to that of sea water (due to some mole-to-gram conversion problem).

It is of interest to study the parameter estimation problem of the fermentation model for the different rates of reaction models with the purposes of control and identification. Online parameter identification can be achieved using recursive state/parameter estimators. For linear systems with normally distributed process and measurement noise, the optimal recursive estimator is the Kalman filter. Estimation for nonlinear systems is considerably more difficult and admits a wider variety of suboptimal solutions. The extended Kalman filter (EKF), unscented Kalman filter (UKF), and the ensemble Kalman filter (EnKF) are implemented using Modelica-Dymosim and Matlab. The fermentation with the different reaction rates is implemented in Modelica and compiled into Dymosim. The parameters are directly estimated using the parameter state-augmented approach and the discrete version of the estimators are implemented in Matlab.

The paper is organized as follows. In the next section, an overview of the fermentation process and its implementation is given. Different kinetic reaction rates for the fermentation process are presented in accordance with biochemical engineering principles. We give a brief introduction of the implementation of the proposed models in Modelica. In section 3, we discuss the problem of parameter estimation of models using recursive nonlinear state/parameter estimation techniques in a Modelica/Matlab setup. The traditional use of the Extended Kalman Filter poses some questions regarding the computation of the Jacobians of the system. In more modern techniques such as the Unscented Kalman Filter, and Monte Carlo techniques such as the Ensemble Kalman Filter, the computation of Jacobians is avoided. Also, these more modern techniques handle nonlinearities in a better way than the Extended Kalman Filter. In particular when these estimation techniques are used for parameter estimation, some of the filter constants need to be carefully tuned, and we discuss this problem. Also, the use of Modelica for "on-line" application such as state estimation poses some particular problems with regards to the efficiency of Modelica implementations; this will be discussed. Finally, we assume that the model of [1] has been fitted well to experimen-

tal data. We then generate fictitious experimental data from the model of Agachi et al., and we illustrate the parameter estimation procedure by fitting the new models to the generated experimental data.

2 Fermentation model

2.1 Description

The nutrients in biochemical reactions are known as substrates. The substrate for the ethanol production process is thus glucose. For the yeast growth process, the substrates are glucose and oxygen. In the sequel we will use symbol S to denote glucose. Since oxygen has a relatively simple chemical formulae, we will not introduce a particular notation for oxygen. Furthermore, we will use symbol P for the main product, which is ethanol, and symbol X for the yeast.

The *original* reaction kinetics given by [1] can be seen in Table 3 with the superscript o for every specie r_j^o .

The fermentation reactor for the production of ethanol is sketched in Fig. 1. Glucose (*substrate* S , sugar) in a water solution is continuously fed to the well stirred reactor; the volumetric feed flow is \dot{V}_i [volume/time]. The reactor contains yeast (microorganisms X), which reacts with substrate to produce ethanol (*product* P). We consider this reaction 1, with kinetic reaction rate r_1 [mass/(volume time)]¹.



Simultaneously, in a second reaction (2), the microorganism breed under the consumption of oxygen to produce more yeast; the kinetic reaction rate is r_2 [mass/(volume time)].



The relationships between the rates of generation r_j [mass/(volume time)] with $j \in \{P, X, S, O_2\}$ can be seen in Table 3. All reaction rates have dimension mass/(volume time). It follows that $r_1 = r_P$ is the mass of ethanol produced per volume and time, etc. Factor Y_{SP} has the meaning of *mass of ethanol (product) produced per mass of glucose (substrate) consumed*. Similar interpretations are valid for Y_{SX} and Y_{OX} .

¹The CO_2 specie is not considered in the expression.

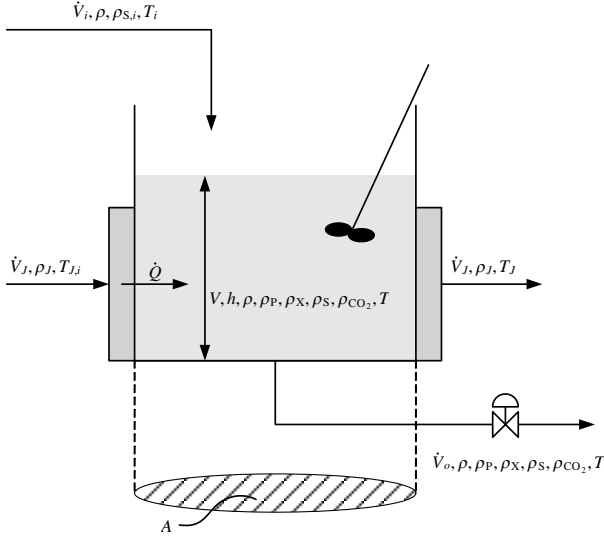


Fig. 1: Sketch of fermentation reactor.

Both in the inlet stream and in the reaction medium, water is dominant such that the density ρ of the mixture can be assumed to be constant. For oxygen, there is an input flow $\dot{m}_{O_2,a}$ in that oxygen is transported from air to dissolved oxygen in the reaction medium,

$$\dot{m}_{O_2,a} = k_{\ell a} (\rho_{O_2}^* - \rho_{O_2}) V, \quad (3)$$

where $k_{\ell a}$ [1/time] depends on the temperature, V is the volume of the reaction medium, ρ_{O_2} is the mass based concentration [mass/volume], and $\rho_{O_2}^*$ is the equilibrium concentration of oxygen in water. $\rho_{O_2}^*$ depends on salts in the mixture. It is assumed that there is no O_2 in the feed water stream. In both reactions 1 and 2, CO_2 is released as a byproduct; here we do not model the carbon dioxide production.

The total mass, species balances, and energy balance for the reactor and the water jacket models are presented in Table 1. The fermentation boundary conditions defined as inputs and outputs are defined in this table. The fermentation model parameters of the *original* model developed in [1] are shown in Table 2.

2.1.1 Fermentation reaction and rates

The *elementary* reaction rate r_1^e for the ethanol production is developed considering the substrate-enzyme interactions, the resulting rate is given by the *Michaelis-Menten kinetics*. Additionally, the presence of ethanol inhibits the ethanol produc-

Tab. 1: Fermentation model.

Reactor total mass and species balances:

$$\frac{d}{dt} m = \dot{m}_i - \dot{m}_o$$

$$\frac{d}{dt} m_j = \dot{m}_{j,i} - \dot{m}_{j,o} + \dot{m}_{j,g}^k \text{ with } j \in \{P, X, S, O_2\}$$

Reactor rates of generation:

$$\dot{m}_{j,g} = r_j^k V \text{ with } j \in \{P, X, S, O_2\}$$

Reactor outputs:

$$\dot{m}_o = k\sqrt{V}$$

$$\dot{m}_{j,o} = \dot{V}_o \rho_j \text{ with } j \in \{P, X, S, O_2\}$$

Reactor inputs:

$$\dot{m}_i = \rho \dot{V}_i$$

$$\dot{m}_{P,i} \equiv 0$$

$$\dot{m}_{X,i} \equiv 0$$

$$\dot{m}_{S,i} = \rho_{S,i} \dot{V}_i$$

$$\dot{m}_{O_2} = \dot{m}_{O_2,a}$$

Oxygen interface transport:

$$\dot{m}_{O_2,a} = k_{\ell a} (\rho_{O_2}^* - \rho_{O_2}) V$$

$$\rho_{O_2}^* = \rho_{O_2,0}^*(T) \exp(-\sum_n I_n - \sum_m S_m)$$

with $n \in \{Na^+, Cl^-, Ca^{+2}, CO_3^{-2}, Mg^{+2}, H^+, OH^-\}$

$$I_n = \frac{1}{2} H_j z_j^2 c_j$$

$$\sum_m S_m = S_S = K_S c_S$$

$$\rho_{O_2,0}^*(T) = \alpha_0 + \alpha_1 T + \alpha_2 T^2 + \alpha_3 T^3$$

Reactor energy balance:

$$\rho \hat{c}_p V \frac{dT}{dt} = \rho \hat{c}_p \dot{V}_i (T_i - T) + \Delta H_{r,2} V r_{O_2}^j - \dot{Q}_{\text{heatex}}$$

Water jacket mass balance:

$$\dot{m}_{J,i} + \dot{m}_{J,o} = 0$$

Water jacket energy balance:

$$\rho_J \hat{c}_{p,J} V_J \frac{dT_J}{dt} = \rho_J \hat{c}_{p,J} \dot{V}_J (T_{J,i} - T_J) + \dot{Q}_{\text{heatex}}$$

Water jacket-reactor heat transfer:

$$\dot{Q}_{\text{heatex}} = U_x A_x (T - T_J)$$

tion rate (inactive enzymes), this effect is also included in this reaction rate. The combined effect is shown in Table 3.

A common simplified model for the effect of competition for active sites yields the *simplified* rate r_1^s , where a specie that competes for an active site and participate in the reaction has the form $\rho_S / (K_{S,1} + \rho_S)$, while a specie that competes for an active site and does not participate in the reaction has the form $1 / (1 + k_{P,1} \rho_P)$.

Another possible model for ethanol production with ethanol inhibition is to notice that $\exp(-k_{P,1} \rho_P) \approx 1 / (1 + k_{P,1} \rho_P)$. This exponential term can be explained by assuming inhibition by ethanol may be caused by *intracellular* mechanisms.

A similar analysis can be done for the reaction rate for the yeast production for the different ap-

Tab. 2: Parameters for the fermentation reactor with original reaction rates.

Reactor/Water jacket parameters:	
$\rho = 1080 \text{ g/l}$	$\Delta\tilde{H}_{r,\text{O}_2} = -518 \text{ kJ/mol}_{\text{O}_2}$
$\rho_J = 1000 \text{ g/l}$	$V_J = 501$
$\hat{c}_p = 4.18 \text{ J/(g }^\circ\text{C)}$	$U_x A_x = 3.6 \text{ E5 J/(h }^\circ\text{C)}$
$\hat{c}_{p,J} = 4.18 \text{ J/(g }^\circ\text{C)}$	$(k_{la})_0 = 38 \text{ h}^{-1}$
Rate of generation parameters:	
$\mu_1 = 1.79 \text{ h}^{-1}$	$K_{S,2} = 1.03 \text{ g/l}$
$\mu_{\text{O}_2} = 0.5 \text{ h}^{-1}$	$K_{\text{O}_2} = 8.86 \text{ mg/l}$
$A_1 = 9.5 \text{ E8 h}^{-1}$	$k_{P,1} = 0.0701/\text{g}$
$A_2 = 2.55 \text{ E33 h}^{-1}$	$k_{P,2} = 0.1391/\text{g}$
$E_{a1}/R = 6.6185 \text{ E3 K}$	$Y_{\text{SX}} = 0.607 \text{ g}_X/\text{g}_S$
$E_{a2}/R = 26.474 \text{ E3 K}$	$Y_{\text{SP}} = 0.435 \text{ g}_P/\text{g}_S$
$K_{S,1} = 1.68 \text{ g/l}$	$Y_{\text{OX}} = 0.970 \text{ g}_X/\text{g}_{\text{O}_2}$
Oxygen interface transport parameters:	
$z_{\text{Na}^+} = +1$	$H_{\text{Mg}^{+2}} = -0.3141/\text{mol}$
$z_{\text{Cl}^-} = -1$	$H_{\text{Ca}^{+2}} = -0.3031/\text{mol}$
$z_{\text{Ca}^{+2}} = +2$	$K_S = 0.1191/\text{mol}$
$z_{\text{CO}_3^{2-}} = -2$	$M_{\text{NaCl}} = 58.44 \text{ g}$
$z_{\text{Mg}^{+2}} = +2$	$M_{\text{MgCl}_2} = 95.21 \text{ g/mol}$
$z_{\text{H}^+} = +1$	$M_{\text{CaCO}_3} = 100.09 \text{ g/mol}$
$z_{\text{OH}^-} = -1$	$M_{\text{O}_2} = 32 \text{ g/mol}$
$H_{\text{Na}^+} = -0.551/\text{mol}$	$M_S = 180.15 \text{ g/mol}$
$H_{\text{Cl}^-} = 0.841/\text{mol}$	$\alpha_0 = 14.16 \text{ mg/l}$
$H_{\text{OH}^-} = 0.941/\text{mol}$	$\alpha_1 = -0.394 \text{ mg/(}^\circ\text{C)}$
$H_{\text{CO}_3^{2-}} = 0.481/\text{mol}$	$\alpha_2 = 7.71 \text{ E}^{-3} \text{ mg/(}^\circ\text{C}^2)$
$H_{\text{H}^+} = -0.771/\text{mol}$	$\alpha_3 = -6.4 \text{ E}^{-5} \text{ mg/(}^\circ\text{C}^3)$

proximations.

The *original* rates are closely related to the developed rates where product inhibition is explained via *intracellular* transport. The *original* model neglects the oxygen dependence of the *intracellular* model and neglects the substrate dependence and the product inhibition. Clearly, when the kinetic rates change their functional form, the parameter/temperature functions change. The different rate reaction rates are shown in Table 3.

2.2 Implementation

In Modelica it is important to implement a good structure to enable easy modification of the models. The core model of the fermentation reactor is the basic volume model, there is where the total mass, species mass balances, and energy balance are defined. This model exchanges heat with the water jacket model through an MSL heat port. It

Tab. 3: Parameters for the fermentation reactor with original reaction rates.

Reaction rates 1:	
$r_1^o = \mu_1 \rho_X \frac{\rho_S}{K_{S,1} + \rho_S} \exp(-k_{P,1} \rho_P)$	
$r_1^e = \mu_1 \rho_X \frac{\rho_S}{K_{S,1} + (1+k_{P,1} \rho_P) + \rho_S}$	
$r_1^s = \mu_1 \rho_X \frac{\rho_S}{K_{S,1} + \rho_S} \frac{1}{1+k_{P,1} \rho_P}$	
$r_1^i = \mu_1 \rho_X \frac{\rho_S}{K_{S,1} + \rho_S} \exp(-k_{P,1} \rho_P)$	
Reaction rates 2:	
$r_2^o = \mu_2 \rho_X \frac{\rho_S}{K_{S,2} + \rho_S} \exp(-k_{P,2} \rho_P)$	
$r_2^e = \mu_2 \rho_X \frac{\rho_S \rho_{\text{O}_2}}{K_{S,2} K_{\text{O}_2} + (1+k_{P,2} \rho_P) + K_{\text{O}_2} \rho_S + \rho_S \rho_{\text{O}_2}}$	
$r_2^s = \mu_2 \rho_X \frac{\rho_S}{K_{S,2} + \rho_S} \frac{\rho_{\text{O}_2}}{K_{\text{O}_2} + \rho_{\text{O}_2}} \frac{1}{1+k_{P,2} \rho_P}$	
$r_2^i = \mu_2 \rho_X \frac{\rho_S}{K_{S,2} + \rho_S} \frac{\rho_{\text{O}_2}}{K_{\text{O}_2} + \rho_{\text{O}_2}} \exp(-k_{P,2} \rho_P)$	
Rates of reactions for P, X, S, O ₂	
$r_P^k = r_1^k$	$k = \{o, e, s, i\}$
$r_X^k = r_2^k$	$k = \{o, e, s, i\}$
$r_S^k = -\frac{1}{Y_{\text{SP}}} r_1^k - \frac{1}{Y_{\text{SX}}} r_2^k$	$k = \{o, e, s, i\}$
$r_{\text{O}_2}^{k*} = -\frac{1}{Y_{\text{OX}}} r_2^{k*}$	$k^* = \{e, s, i\}$
$r_{\text{O}_2}^o = -\frac{1}{Y_{\text{OX}}} \mu_{\text{O}_2} \rho_X \frac{\rho_{\text{O}_2}}{K_{\text{O}_2} + \rho_{\text{O}_2}}$	

also has a chemical port (i.e. *intensive variables*: temperature and mass concentration vector; and *extensive variables*: mass flow rates vector and heat flow rate) that connects with the rate of generation replaceable model and the oxygen transport model, and two thermofluid ports (i.e. *intensive variables*: pressure, specific enthalpy, and mass fraction vector; and *extensive variables*: enthalpy flow rate vector, mass flow rate vector, and total mass flow rate) to connect the basic volume with the incoming mass flow rate in and the outgoing mass flow rate out of the model. The basic volume is then connected to the water jacket model, to the oxygen transport model, to the rate of generation replaceable model as shown in Fig. 2.

The four different reaction kinetic rates (i.e. *original*, *elementary*, *simplified*, and *intracellular*) are implemented using a replaceable component. A common set of parameters and equations are defined in a partial model called rate of generation. Specifics of every reaction rate model are defined separately in each model that inherits the rate of generation partial model. The heat of reaction is also defined in these models. The water jacket model uses two MSL flow ports.

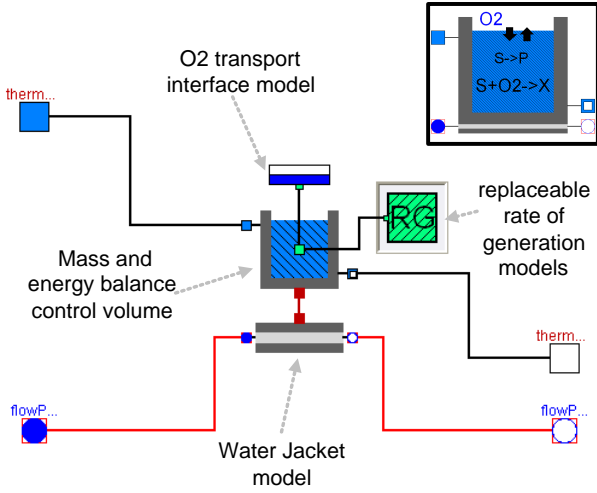


Fig. 2: Dymola diagram layout of the fermentation reactor component.

3 Nonlinear estimators

3.1 Description

The ethanol fermentation reactor model can be written in the general discrete nonlinear state space form:

$$\begin{aligned} x_k &= f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}) \\ y_k &= h_k(x_k, v_k) \end{aligned} \quad (4)$$

where $f_{k-1} : \mathbb{R}^{n_x+n_u+n_w} \rightarrow \mathbb{R}^{n_x}$ is the discrete state function, $x_k \in \mathbb{R}^{n_x}$ is the discrete state vector, $u_{k-1} \in \mathbb{R}^{n_u}$ is the discrete input, $w_{k-1} \in \mathbb{R}^{n_w}$ is the discrete process noise vector, $h_k : \mathbb{R}^{n_x+n_v} \rightarrow \mathbb{R}^{n_y}$ is the discrete output function, $v_k \in \mathbb{R}^{n_v}$ is the discrete measurement noise vector, $y_k \in \mathbb{R}^{n_y}$ is the output vector, and k is the time index. The noise vector sequences $\{w_{k-1}\}$ and $\{v_k\}$ are assumed Gaussian, white, zero-mean, uncorrelated, and have the known covariance matrices $Q_k \in \mathbb{R}^{n_x \times n_w}$ and $R_k \in \mathbb{R}^{n_y \times n_v}$.

3.2 Augmented states

The augmented state space approach can be directly used to simultaneously solve the state and the parameter estimation problem (e.g. see [4]). An augmented state space representation is formulated by adding the vector of parameters to be estimated $\theta_k \in \mathbb{R}^{n_\theta \times 1}$ as new states:

Tab. 4: EKF algorithm.

<p>Initialization:</p> $\hat{x}_{0 0} \sim \mathcal{N}(\bar{x}_0, P_0)$ $P_{0 0} = P_0$ <p>for $k = 1, 2, \dots$</p> <p>Propagation step:</p> <p>(a priori covariance estimate)</p> $F_{k-1} = \left. \frac{\partial f_{k-1}}{\partial x_{k-1}} \right _{\hat{x}_{k-1 k-1}} \quad L_{k-1} = \left. \frac{\partial f_{k-1}}{\partial w_{k-1}} \right _{\hat{x}_{k-1 k-1}}$ $P_{k k-1} = F_{k-1}P_{k-1 k-1}F_{k-1}^T + L_{k-1}Q_{k-1}L_{k-1}^T$ <p>(a priori state-output estimate)</p> $\hat{x}_{k k-1} = f_{k-1}(\hat{x}_{k-1 k-1}, u_{k-1}, 0)$ $\hat{y}_{k k-1} = h_k(\hat{x}_{k k-1}, 0)$ <p>Measurement update:</p> <p>(Kalman gain calculation)</p> $H_k = \left. \frac{\partial h_k}{\partial u_k} \right _{\hat{x}_{k k-1}} \quad M_k = \left. \frac{\partial h_k}{\partial v_k} \right _{\hat{x}_{k k-1}}$ $K_k = P_{k k-1}H_k^T(H_kP_{k k-1}H_k^T + M_kR_kM_k^T)^{-1}$ <p>(a posteriori state-covariance estimate)</p> $\hat{x}_{k k} = \hat{x}_{k k-1} + K_k(y_k - \hat{y}_{k k-1})$ $P_{k k} = (I - K_kH_k)P_{k k-1}$
--

$$\begin{bmatrix} x_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}^{(x)}) \\ \theta_{k-1} + T_s w_{k-1}^{(\theta)} \end{bmatrix} \quad (5)$$

$$y_k = h_k(x_k, v_k) \quad (6)$$

where T_s is the sampling time step, $w_{k-1}^{(x)} \in \mathbb{R}^{n_w^{(x)}}$ is the process noise vector that affects the original states, and $w_{k-1}^{(\theta)} \in \mathbb{R}^{n_w^{(\theta)}}$ is the process noise vector that affects the added parameter states. The noise vector sequences $\{w_{k-1}\}$ and $\{v_k\}$ are assumed Gaussian, white, zero-mean, uncorrelated, and have the known covariance matrices $Q_k \in \mathbb{R}^{(n_x+n_\theta) \times (n_w^{(x)}+n_w^{(\theta)})}$ and $R_k \in \mathbb{R}^{n_y \times n_v}$

$$\begin{aligned} w_k &\sim \mathcal{N}(0, \text{blkdg}(Q_k^{(x)}, Q_k^{(\theta)})) \\ v_k &\sim \mathcal{N}(0, R_k) \end{aligned}$$

During the propagation step, the augmented states corresponding to parameters θ_k are considered equal to the previous time step θ_{k-1} with some additive process noise $w_{k-1}^{(\theta)}$. If it is assumed that the parameters do not change at all, then there is no process noise vector $w_{k-1}^{(\theta)}$, but for the more general case of time-varying parameters (e.g. fouling, etc.), the value of $Q_k^{(\theta)}$ will be given by the admissible range of variation of θ_k . During

the measurement update step the parameter values are corrected.

For notational simplicity in the estimators algorithms that follow, the augmented state vector is referred to as x_k , the state augmented function (5) is referred to as f_{k-1} , and the augmented process noise vector is referred to as w_{k-1} .

Tab. 5: UKF algorithm.

<p>Initialization:</p> $L = n_x + n_w + n_v, \lambda = \alpha^2(L + \kappa) - L$ $\gamma = \sqrt{2L + \lambda}, \Lambda_m^0 = \lambda/(\lambda + L)$ $\Lambda_c^0 = \lambda/(\lambda + L) + (1 - \alpha^2 + \beta)$ <p>for $i = 1, 2, \dots, 2L$</p> $\Lambda_m^i = (2(\lambda + L))^{-1}, \Lambda_c^i = \Lambda_m^i$ $\hat{x}_{0 0} \sim \mathcal{N}(\bar{x}_0, P_0)$ $P_{0 0} = P_0$ <p>for $k = 1, 2, \dots$</p> <p>Propagation step:</p> <p>(sigma points propagation)</p> $\tilde{P}_{k-1 k-1} = \text{blkdiag}(P_{k-1 k-1}, Q_k, R_k)$ $\tilde{x}_{k-1 k-1}^0 = [(\hat{x}_{k-1 k-1})^T, \mathbf{0}_{1 \times n_w}, \mathbf{0}_{1 \times n_v}]^T$ <p>for $i = 1, 2, \dots, L$</p> $\tilde{x}_{k-1 k-1}^i = \hat{x}_{k-1 k-1}^0 + \gamma \text{chol}(\tilde{P}_{k-1 k-1}, i)$ $\tilde{x}_{k-1 k-1}^{i+L} = \hat{x}_{k-1 k-1}^0 - \gamma \text{chol}(\tilde{P}_{k-1 k-1}, i + L)$ $\tilde{x}_{k k-1}^{(x)i} = f_{k-1}(\tilde{x}_{k-1 k-1}^i, u_{k-1}, \tilde{x}_{k-1 k-1}^{(w)i})$ $\tilde{y}_{k k-1}^i = h_k(\tilde{x}_{k k-1}^{(x)i}, \tilde{x}_{k-1 k-1}^{(v)i})$ <p>(a priori state-output estimate)</p> $\hat{x}_{k k-1} = \sum_{i=0}^{2L} \Lambda_m^i \tilde{x}_{k k-1}^{(x)i}$ $\hat{y}_{k k-1} = \sum_{i=0}^{2L} \Lambda_m^i \tilde{y}_{k k-1}^i$ <p>(a priori state covariance estimate)</p> $\tilde{e}_{x,k k-1}^i = (\tilde{x}_{k k-1}^{(x)i} - \hat{x}_{k k-1})$ $P_{k k-1} = \sum_{i=0}^{2L} \Lambda_c^i (\tilde{e}_{x,k k-1}^i)(\tilde{e}_{x,k k-1}^i)^T$ <p>Measurement update:</p> <p>(Kalman gain calculation)</p> $\tilde{e}_{y,k k-1}^i = (\tilde{y}_{k k-1}^i - \hat{y}_{k k-1})$ $P_y = \sum_{i=0}^{2L} \Lambda_c^i (\tilde{e}_{y,k k-1}^i)(\tilde{e}_{y,k k-1}^i)^T$ $P_{xy} = \sum_{i=0}^{2L} \Lambda_c^i (\tilde{e}_{x,k k-1}^i)(\tilde{e}_{y,k k-1}^i)^T$ $K_k = P_{xy} P_y^{-1}$ <p>(a posteriori state-covariance estimate)</p> $\hat{x}_{k k} = \hat{x}_{k k-1} + K_k (y_k - \hat{y}_{k k-1})$ $P_{k k} = P_{k k-1} - K_k P_y K_k^T$

3.3 Nonlinear Recursive Estimators

The nonlinear estimation problem can be formulated as a recursive Bayesian estimation problem with a propagation and a measurement update

step. This is the optimal way of predicting a state probability density function (pdf) $p(x_k)$ for any system in state space representation with process and measurement noise².

Tab. 6: EnKF algorithm.

<p>Initialization:</p> <p>(initial ensemble)</p> <p>for $i = 1, 2, \dots, N$</p> $x_{0 0}^i \sim \mathcal{N}(\bar{x}_0, P_0)$ <p>for $k = 1, 2, \dots$</p> <p>Propagation step:</p> <p>(ensemble propagation)</p> <p>for $i = 1, 2, \dots, N$</p> $x_{k k-1}^i = f_{k-1}(x_{k-1 k-1}^i, u_{k-1}, w_{k-1}^i)$ $y_{k k-1}^i = h_k(x_{k k-1}^i, v_{k-1}^i)$ <p>(estimated state-output propagation)</p> $\hat{x}_{k k-1} = (N)^{-1} \sum_{i=1}^N x_{k k-1}^i$ $\hat{y}_{k k-1} = (N)^{-1} \sum_{i=1}^N y_{k k-1}^i$ <p>(covariance calculation)</p> $e_{x,k k-1}^i = (x_{k k-1}^i - \hat{x}_{k k-1})$ $P_{k k-1} = (N - 1)^{-1} \sum_{i=1}^N (e_{x,k k-1}^i)(e_{x,k k-1}^i)^T$ <p>Measurement update:</p> <p>(Kalman gain calculation)</p> $e_{y,k k-1}^i = (y_{k k-1}^i - \hat{y}_{k k-1})$ $P_y = (N - 1)^{-1} \sum_{i=1}^N (e_{y,k k-1}^i)(e_{y,k k-1}^i)^T$ $P_{xy} = (N - 1)^{-1} \sum_{i=1}^N (e_{x,k k-1}^i)(e_{y,k k-1}^i)^T$ $K_k = P_{xy} P_y^{-1}$ <p>(state-out-covariance update)</p> $x_{k k}^i = x_{k k-1}^i + K_k ((y_k + v_k^i) - y_{k k-1}^i)$ $\hat{x}_{k k} = (N)^{-1} \sum_{i=1}^N x_{k k}^i$ $P_{k k} = P_{k k-1} - K_k P_y K_k^T$

Assuming that the initial state pdf $p(x_0)$, the process noise pdf $p(w_{k-1})$, and the measurement noise pdf $p(v_k)$ are known, a recursive solution of the estimation problem can be found using first the Chapman-Kolmogorov equation to calculate the *a priori* pdf for the state x_k based on the previous measurement y_{k-1} (propagation step)

$$p(x_k|y_{k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|y_{k-1})dx_{k-1} \quad (7)$$

where $p(x_k|x_{k-1})$ can be calculated from the state function f_{k-1} and the pdf of the process noise w_k .

Secondly, the Bayes rule to update the pdf of the state x_k with the new measurement y_k (measurement update) is

²Markov process of order one.

$$p(x_k|y_k) = \frac{p(y_k|x_k)p(x_k|y_{k-1})}{\int p(y_k|x_k)p(x_k|y_{k-1})dx_k} \quad (8)$$

where $p(y_k|x_k)$ is available from our knowledge of the output function h_k and the pdf of v_k , and $p(x_k|y_{k-1})$ is known from (7). Although the initial state pdf $p(x_0)$, the process noise pdf $p(w_{k-1})$, and the measurement noise pdf $p(v_k)$ are needed to solve the recursive Bayesian estimation, no specific statistical distribution is required.

The recursive relations (7) and (8) used to calculate the *a posteriori* pdf $p(x_k|y_k)$ are a conceptual solution and only for very specific cases can these be solved analytically. In general, approximations are required for practical problems. Three main groups of suboptimal techniques with significant performance and computational cost differences are used to approximate the recursive Bayesian estimation problem: the classical nonlinear extension of the Kalman filter (EKF), the Unscented Kalman filter (UKF), and the Ensemble Kalman filter (EnKF) approaches.

3.4 Extended Kalman Filter (EKF)

The discrete EKF is probably the most used sequential nonlinear estimator nowadays. It was originally developed as a nonlinear extension by Schmidt [5] of the seminal work of Kalman [6]. Based on the Kalman filter, it assumes that the statistical distribution of the state vector remains Gaussian after every time step³ so it is only necessary to propagate and update the mean and covariance of the state random variable x_k . The main concept is that the estimated state (i.e. estimated mean of x_k) is sufficiently close to the true state (i.e. true mean of x_k) so the nonlinear state/output model equations can be linearized by a truncated first-order Taylor series expansion around the previously estimated state.

The discrete algorithm is given in table 4. In general, this algorithm works for many practical problems, but no general convergence or stability conditions can be established⁴ and its final performance will depend on the specific case study. For highly nonlinear models with unknown initial conditions, the EKF assumptions may prove to be poor and the filter may fail or have a poor per-

³this assumption is in general not true for nonlinear systems.

⁴except for some special cases [7].

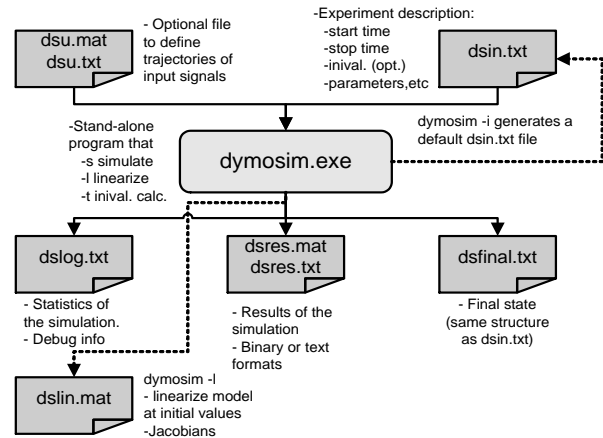


Fig. 3: Dymosim and related input and output files.

formance. The main tuning parameters are the estimator covariance matrices Q_k and R_k .

3.5 Unscented Kalman Filter (UKF)

The unscented Kalman filter was originally developed by Julier and Uhlman [8, 9, 10, 11].

In the unscented Kalman filters, instead of approximating the nonlinear state/output functions, it is the probability distribution that is approximated. Basically, a set of points, called sigma points, are generated to match the state mean and state covariance of the probability distribution of the previously estimated state, then they are propagated through the nonlinear function. The projected points are used to approximate the first two moments (i.e. the *a priori* estimated state and state covariance) that are necessary during the measurement update step. This filter normally outperforms the previously presented EKF. Its more general form has a higher computational cost but it does not require the calculation of any Jacobian matrices (i.e. derivatives). The algorithm is given in table 5.

The tuning parameters of the UKF are also the estimator process and measurement noise covariance matrices, and the scalar parameters $\{\alpha, \kappa, \beta\}$: α determines the spread of the sigma points around the previous estimate, and the β value depends on the type of distribution assumed (for more details about their values see [11]).

3.6 Ensemble Kalman Filter (EnKF)

The EnKF uses an ensemble (i.e. particle set) during the propagation step, but the classical Kalman

measurement update equations (instead of using the resampling with replacement approach of the particle filters) during the measurement update step. The covariances matrices P_{xy} and P_y obtained from the propagation of the ensemble elements through the nonlinear state-space are used to calculate the Kalman gain K_k . The *a posteriori* ensemble is calculated from the Kalman gain matrix and an artificially generated measurement particle set that is normally distributed with mean equal to the current measurement y_k and covariance equal to R_k . The *a posteriori* ensemble is used to calculate the *a posteriori* state and covariance estimate, and it is used for the next filter iteration of the algorithm. For details about the algorithm, see table 6. The EnKF was originally developed in [12] to overcome the curse of dimensionality in large scale problems (i.e. weather data assimilation). It is suggested in the literature [13] that ensembles (i.e. particle sets) of 50 to 100 are often adequate for systems with thousands of states, but no conclusive work has been done on this.

Besides the estimator process and measurement noise covariance matrices, the other tuning parameter for this filter is the number of ensemble elements.

3.7 Implementation

The fermentation model is written in Modelica and compiled in Dymola into a stand-alone executable file called Dymosim. The different estimators are implemented in Matlab from where Dymosim is sequentially called during the propagation step to project the state vector (i.e. integrate over the sampling time) in the estimator algorithms. The parameter state vector θ_k is directly propagated within the Matlab code so the original model does not need to be modified to include the parameter dynamic equations.

Within the Modelica model the input vector u_k , the process noise input vector w_k , and the parameter input vector θ_k must be defined. This can be done in the following way at the top level of the model:

```

model fermentation
...
input Real u_ul; // define model inputs
input Real u_wl; // define noise inputs
input Real u_pl; // define param. inputs
...
parameter Real p_ul;
parameter Real p_wl;
parameter Real p_pl;
parameter Real p_il;
equation
fluidBCv.u[1]=u_ul+p_ul;
reactor.basicVol.w[1]=u_wl+p_wl;
reactor.RG.p_mul=u_pl+p_pl;
reactor.i_rho[1]=p_il;

end fermentation;

```

Additionally, the discrete EKF estimator requires the calculation of the discrete Jacobians $F_{k-1}, L_{k-1}, H_k, M_k$. This can be done calculating a linearized model around the previous state estimate defined by the operating point $op = [x_{k-1}^T, u_{k-1}^T, 0, \theta_{k-1}^T]^T$ with the following Matlab code:

```
eval( ['! dymosim ', '-l ', 'dsin.txt' ] );
```

In the file “dsin.txt” (see Fig. 3) the operating point is defined using parameters and the initial state for every iteration. The calculated linearized model is written in the file “dslin.mat” and then it can be loaded into Matlab using the Dymola add-on function `tloadlin` which loads the matrices A, B, C, D and the string vectors `uname`, `yname`, and `xname`. These matrices correspond to

$$\begin{aligned}
 A &= \left. \frac{\partial f(x,u,w)}{\partial x} \right|_{op} \\
 B &= \left[\left. \frac{\partial f(x,u,w)}{\partial u} \right|_{op}, \left. \frac{\partial f(x,u,w)}{\partial w} \right|_{op}, \left. \frac{\partial f(x,u,w)}{\partial \theta} \right|_{op} \right] \\
 C &= \left. \frac{\partial h(x,u)}{\partial x} \right|_{op} \\
 D &= \left. \frac{\partial h(x,u)}{\partial u} \right|_{op}
 \end{aligned}$$

The parameter augmented state space discrete jacobians are approximated from the A, B, C, D matrices

$$\begin{aligned}
 A^e &= \begin{bmatrix} A & B(:,n_u+n_w+1:end) \\ 0_{n_p \times n_x} & 0_{n_p \times n_p} \end{bmatrix} \\
 F_{k-1} &= \left. \frac{\partial f_{k-1}}{\partial x_{k-1}} \right|_{op} \approx \exp(A^e \Delta t) \\
 B^e &= \begin{bmatrix} B(:,n_u+1:end) & 0_{n_x \times n_p} \\ 0_{n_p \times n_x} & 1_{n_p \times n_p} \end{bmatrix} \\
 L_{k-1} &= \left. \frac{\partial f_{k-1}}{\partial w_{k-1}} \right|_{op} \\
 &\approx [I \Delta t + \frac{1}{2!} A^e \Delta t^2 + \frac{1}{3!} A^{e2} \Delta t^3 + \dots] B^e \\
 H_k &= [C \ 0_{n_y \times n_x}] \\
 M_k &= D
 \end{aligned}$$

where n_u is the input vector dimension, n_w is the process noise vector dimension, and so on. For notation simplicity, the matrices in the previous equations use Matlab notation.

4 Results

Due to the lack of experimental measurements, simulated data sets from the model with the original kinetic reaction rates are generated. The system model is simulated for 1000 h and data samples are collected every 1 h. Because the transient response is relevant to parameter identification, step-like input sequences with high frequency content are used (see Fig.4). The initial state vector for the fermentation model is

$$\begin{aligned} x_0 &= [\dot{V}, \rho_P, \rho_X, \rho_S, \rho_{O_2}, T, T_J]^T \\ &= [1000, 12.9, 0.9, 28.6, 3.9, 30.4, 26.9]^T \end{aligned}$$

The system model process and measurement noise vector sequences $\{w_{k-1}\}$ and $\{v_k\}$ are Gaussian, white, zero-mean, uncorrelated, and have constant covariance matrices

$$\begin{aligned} Q_k &= \text{blkdiag}(Q_k^{(x)}, Q_k^{(\theta)}) \\ Q_k^{(x)} &= \text{diag}([1000, 15, 2, 100, 5, 35, 30]) * 1E-7 \\ Q_k^{(\theta)} &= \text{diag}([1, 1, 1, 1, 1, 1, 1]) * 1E-7 \\ R_k &= \text{diag}([15, 2, 100, 5, 35, 30]) * 2E-3 \end{aligned}$$

A subset of 8 parameters $\theta = [\mu_1, K_{S,1}, K_{S,2}, k_{P,1}, k_{P,2}, Y_{SP}, Y_{SX}, Y_{OX}]^T$ is estimated for every estimator (i.e., the EKF, the UKF, and the EnKF) using every reaction rate model (i.e., the *original*, the *elementary*, the *simple*, and the *intracellular* reaction kinetic models). The initial parameter values for every reaction rate model are adjusted to ensure that all simulation results give the same steady state values at initial time $t = 0$. The estimators inputs are equal to the system model inputs $u = [\dot{V}_{J,i}, \rho_{S,i}]^T$, and the measured outputs are $y = [\rho_P, \rho_X, \rho_S, \rho_{O_2}, T, T_J]^T$ (see Fig.4). The estimators are simulated for 1000 h with a sampling time of 1 h.

The estimators' initial state vectors are drawn from a normal distribution with mean and covariance equal to

$$\begin{aligned} \hat{x}_{0|0} &\sim \mathcal{N}([\bar{x}_0^T, \bar{\theta}_0^T]^T, \text{blkdiag}(P_0^{(x)}, P_0^{(\theta)})) \\ \bar{x}_0 &= [990, 13.9, 0.8, 27.6, 4.9, 27.4, 24.9]^T \\ \bar{\theta}_0 &= [1.49, 1.48, 1.23, 7.3, 1.19, 5.07, 4.55, 9.3]^T \\ P_0^{(x)} &= \text{diag}((0.125 * \bar{x}_{0|0}) .^2) \\ P_0^{(\theta)} &= \text{diag}((0.125 * \bar{\theta}_{0|0}) .^2) \end{aligned}$$

The UKF parameters are $\{\alpha, \kappa, \beta\} =$

$\{1E-3, 0, 0\}$, and the EnKF is evaluated for an ensemble of $N = 100$ elements. The estimators process and measurement noise sequences are Gaussian, white, zero-mean, uncorrelated, and have constant covariance matrices equal to the system model.

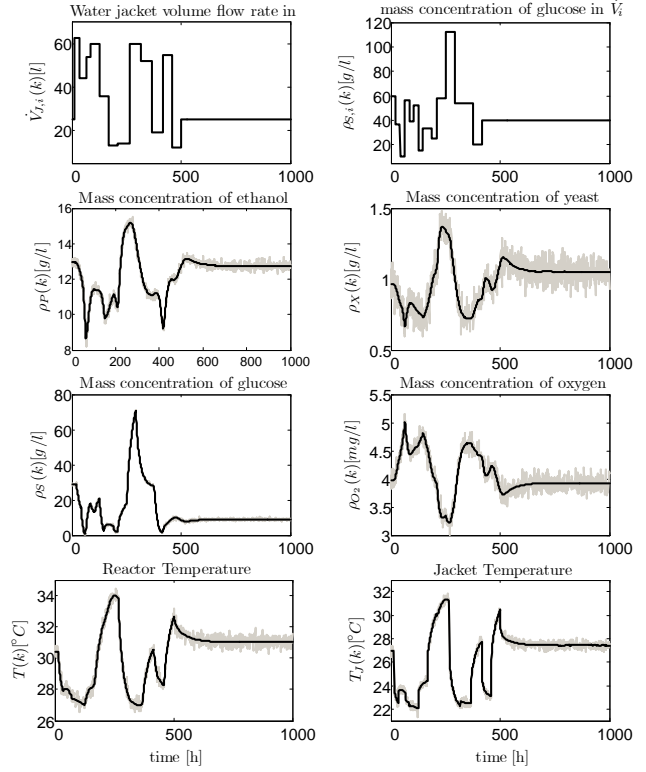


Fig. 4: Process inputs ($\dot{V}_{J,i}, \rho_{S,i}$), and measured outputs ($\rho_P, \rho_X, \rho_S, \rho_{O_2}, T, T_J$) with measurement noise (grey line) and without it (black line).

Every different reaction rate is evaluated for every estimator using 50 Monte Carlo simulations. As a general notation, consider an ensemble $\{x_j^i(k)\}$ where i indicates the realization, j the state/parameter, and k the time index. The ensemble average (over the realizations) is denoted $\langle x_j^i(k) \rangle$:

$$\langle x_j^i(k) \rangle \triangleq \frac{\sum_i x_j^i(k)}{n_{\text{simul}}}$$

where n_{simul} is the number of realizations.

For every estimator with the different reaction rates two performance values (averaged over the number of Monte Carlo simulations) are calculated for each estimated parameter j : the *averaged estimated parameter* for every time index k that is used to evaluate the parameter estimation bias wrt. the true parameter value $\langle \hat{\theta}_j^i(k) \rangle$, and the *averaged absolute estimated parameter error* de-

defined as $\langle |e_{\hat{\theta}_j^i}^i(k)| \rangle \triangleq \langle |\hat{\theta}_j^i - \hat{\theta}_j^i(k)| \rangle$ for every time index k . This second performance value is used to evaluate the convergence and consistency of every estimator.

The Monte Carlo averaged performance of the estimators using the *original* reaction rate model is shown in Fig. 5. The averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ converge to the true parameters for all the parameters except for the slightly biased \hat{Y}_{SP} estimate and the more biased \hat{Y}_{SX} estimate. In Fig. 5 column (b), the averaged estimated parameter errors for the parameters $\{\hat{k}_{P,2}, \hat{Y}_{SP}, \hat{Y}_{O_2}\}$ converge at a faster rate than for the other estimated parameters $\{\hat{\mu}_1, \hat{K}_{S,1}, \hat{K}_{S,2}, \hat{k}_{P,1}, \hat{Y}_{SX}\}$. The EKF and the UKF have comparable averaged estimated parameters, while the EnKF has slightly biased averaged estimated parameters. The best performance wrt. the averaged absolute estimated parameter error $\langle |e_{\hat{\theta}_j^i}^i(k)| \rangle$ is achieved for the EKF followed by the UKF and the EnKF.

The Monte Carlo averaged performance of the estimators using the *elementary* reaction rate model is shown in Fig. 6. It can be seen that the averaged estimated parameters no longer converge to the true parameters of the *original* rate model used in the system model simulations. It is to be expected that some of the parameters will be time-varying to compensate for the different kinetic rates (between the system and the estimator kinetic rate models) and, in this way, keep a good state estimation performance besides their differences. For this case, the averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ are considered as an unbiased estimate of the true (possibly time-varying) parameters. The averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ take different shapes over time depending on the specific estimator evaluated. In Fig. 6 column (b), the averaged absolute estimated parameter errors $\langle |e_{\hat{\theta}_j^i}^i(k)| \rangle$ for the parameters $\{\hat{k}_{P,1}, \hat{k}_{P,2}, \hat{Y}_{SX}, \hat{Y}_{SP}\}$ and the EKF diverge while the UKF achieves the best performance followed by the EnKF. It is then reasonable to consider that the averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ that correspond to the UKF are the best estimate of the true parameters $\theta_j(k)$ for this estimator reaction rate model.

The Monte Carlo averaged performance of the estimators using the *simplified* reaction rate model is shown in Fig. 7. As for the *elementary* case, the averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ are considered as an unbiased estimate of the true (possibly time-varying) parameters. The averaged es-

timated parameters $\langle \hat{\theta}_j(k) \rangle$ have similar values for the EKF and the UKF and slightly different for the EnKF. In Fig. 7 column (b), the lowest averaged absolute estimated parameter errors $\langle |e_{\hat{\theta}_j^i}^i(k)| \rangle$ are achieved for the EKF, closely followed by the UKF performance. For all the estimators the averaged absolute estimated parameter errors decrease over time.

The Monte Carlo averaged performance of the estimators using the *intracellular* reaction rate model is shown in Fig. 8. As for the *elementary* and *simplified* cases, the averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ are considered as an unbiased estimate of the true (possibly time-varying) parameters. The averaged estimated parameters $\langle \hat{\theta}_j(k) \rangle$ have similar values for the EKF and the UKF and slightly different for the EnKF. In Fig. 8 column (b), the averaged absolute estimated parameter errors $\langle |e_{\hat{\theta}_j^i}^i(k)| \rangle$ decrease over time for all the parameters and estimators, except for the estimated parameter \hat{Y}_{SX} with the EnKF.

In Table 7 the different reaction rate models are evaluated for each filter using the normalized mean RMSE defined as

$$\overline{\text{RMSE}}(x) = \sum_j^{n_x} \frac{\sum_i^{n_{\text{simul}}} \sqrt{\frac{\sum_k^{n_t} (\hat{x}_j(k) - x_j^{\text{true}}(k))^2}{n_t}}}{\max(x_j^{\text{true}}) - \min(x_j^{\text{true}})}$$

Tab. 7: Normalized mean RMSE for the estimated state x and parameter θ vectors. The best results for every case is indicated by parentheses.

RMSE(.)		EKF	UKF	EnKF
Original	x	9.11E-2	9.13E-2	(8.44E-2)
	θ	(1.22)	1.74	1.95
Elementary	x	2.09E-1	1.00E-1	(9.35E-2)
	θ	5.57E-1	(2.73E-1)	7.54E-1
Simplified	x	8.36E-2	(8.05E-2)	8.77E-2
	θ	(3.21E-1)	3.56E-1	3.28E-1
Intracellular	x	8.94E-2	1.05E-1	(8.60E-2)
	θ	(3.98E-1)	5.69E-1	4.71E-1

5 Conclusions

The recursive parameter estimation problem is analyzed for an ethanol fermentation process with different reaction rate models. The model is implemented in Modelica and three nonlinear estimators are evaluated using the compiled Modelica model (Dymosim) with Matlab. Implementation details (e.g. how to calculate Jacobians, defined noise inputs, etc.) are presented.

Some relevant model parameters are estimated using the EKF, the UKF, and the EnKF from sim-

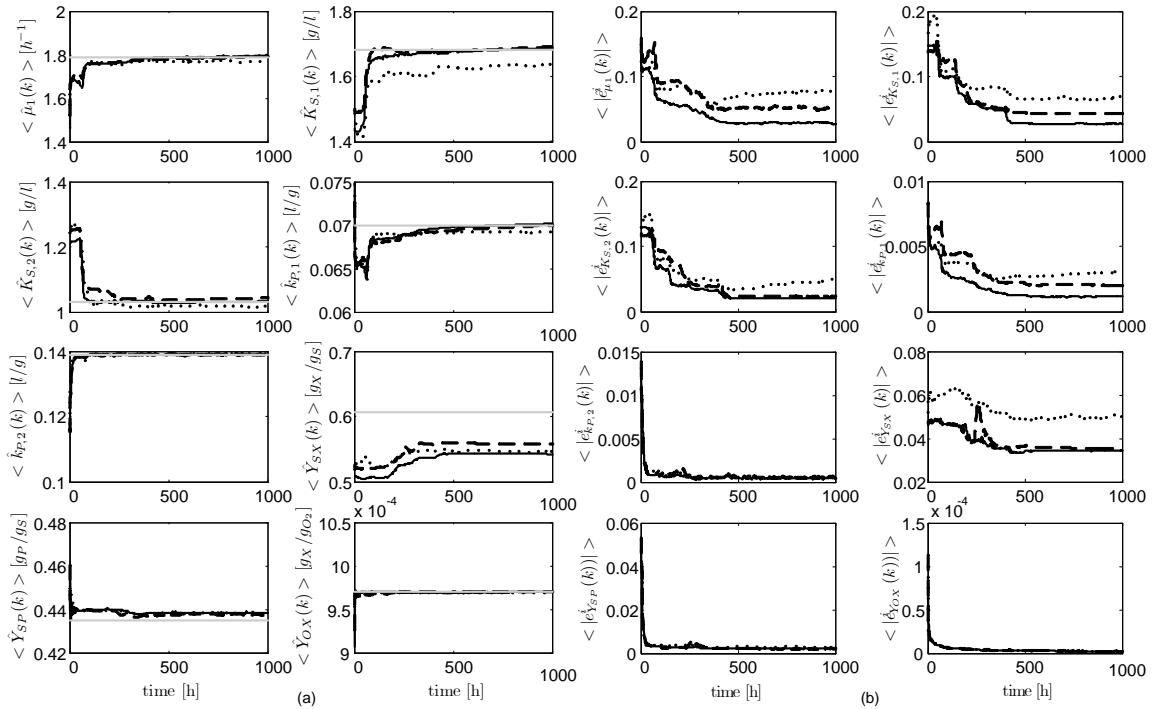


Fig. 5: *Original* kinetic rate model parameter estimation results, averaged over 50 Monte Carlo simulations for the EKF (black solid line), the UKF (black dash line), and the EnKF (black dotted line): (a) mean parameter estimates $\langle \hat{\theta}_j(k) \rangle$ for every time index k and true parameters θ_j (grey solid line); (b) mean absolute estimated parameter error, $\langle |e_{\theta_j}^i| \rangle = \langle |\hat{\theta}_j^i - \langle \hat{\theta}_j(k) \rangle| \rangle$ for every time index k .

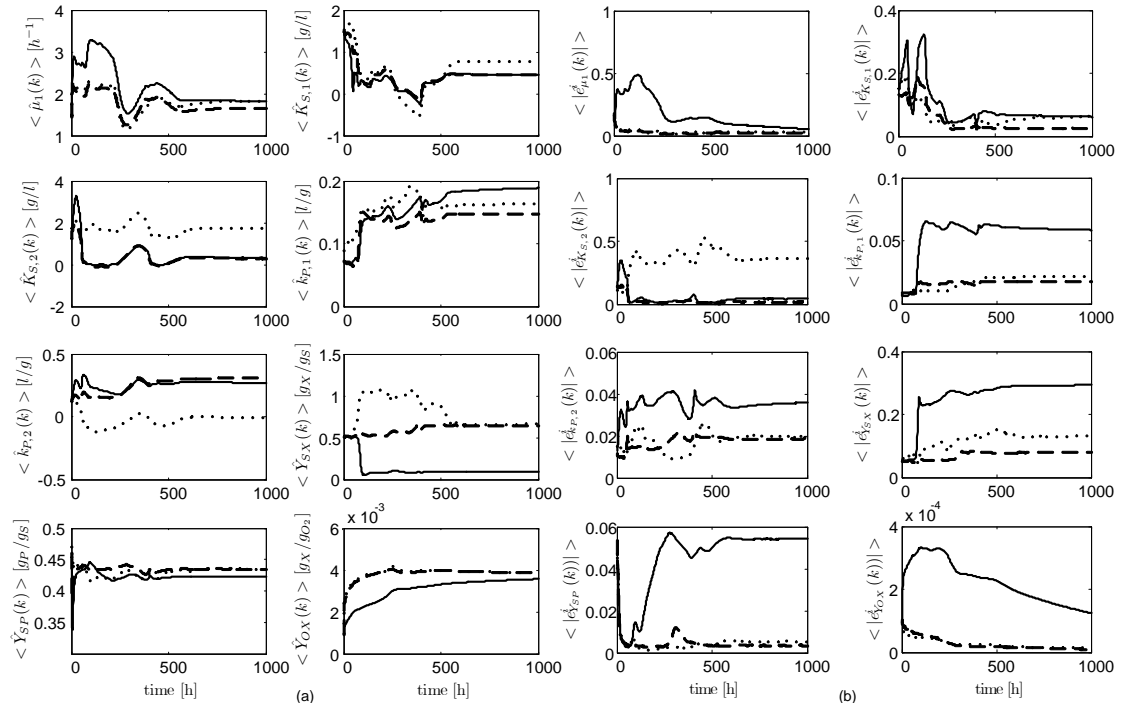


Fig. 6: *Elementary* kinetic rate model parameter estimation results, averaged over 50 Monte Carlo simulations for the EKF (black solid line), the UKF (black dash line), and the EnKF (black dotted line): (a) mean parameter estimates $\langle \hat{\theta}_j(k) \rangle$ for every time index k ; (b) mean absolute estimated parameter error, $\langle |e_{\theta_j}^i| \rangle = \langle |\hat{\theta}_j^i - \langle \hat{\theta}_j(k) \rangle| \rangle$ for every time index k .

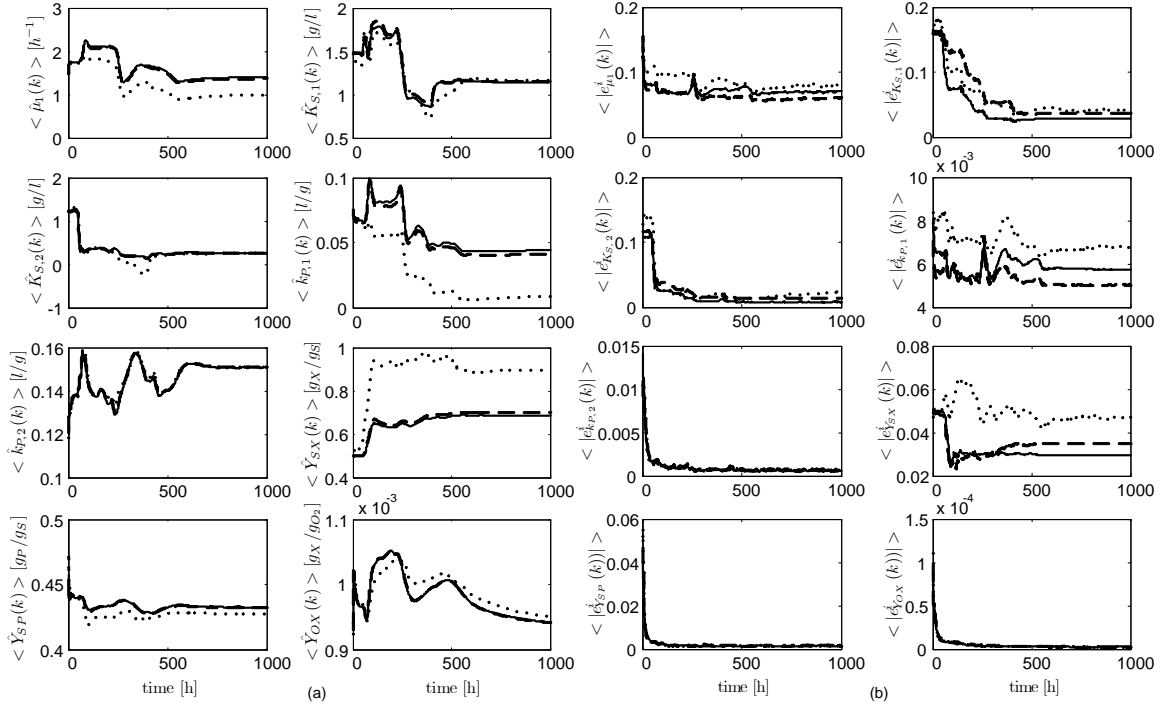


Fig. 7: *Simplified* kinetic rate model parameter estimation results, averaged over 50 Monte Carlo simulations for the EKF (black solid line), the UKF (black dash line), and the EnKF (black dotted line): (a) mean parameter estimates $\langle \hat{\theta}_j^i(k) \rangle$ for every time index k; (b) mean absolute estimated parameter error, $\langle |e_{\theta_j^i}^i| \rangle = \langle |\hat{\theta}_j^i - \hat{\theta}_j^i(k)| \rangle$ for every time index k.

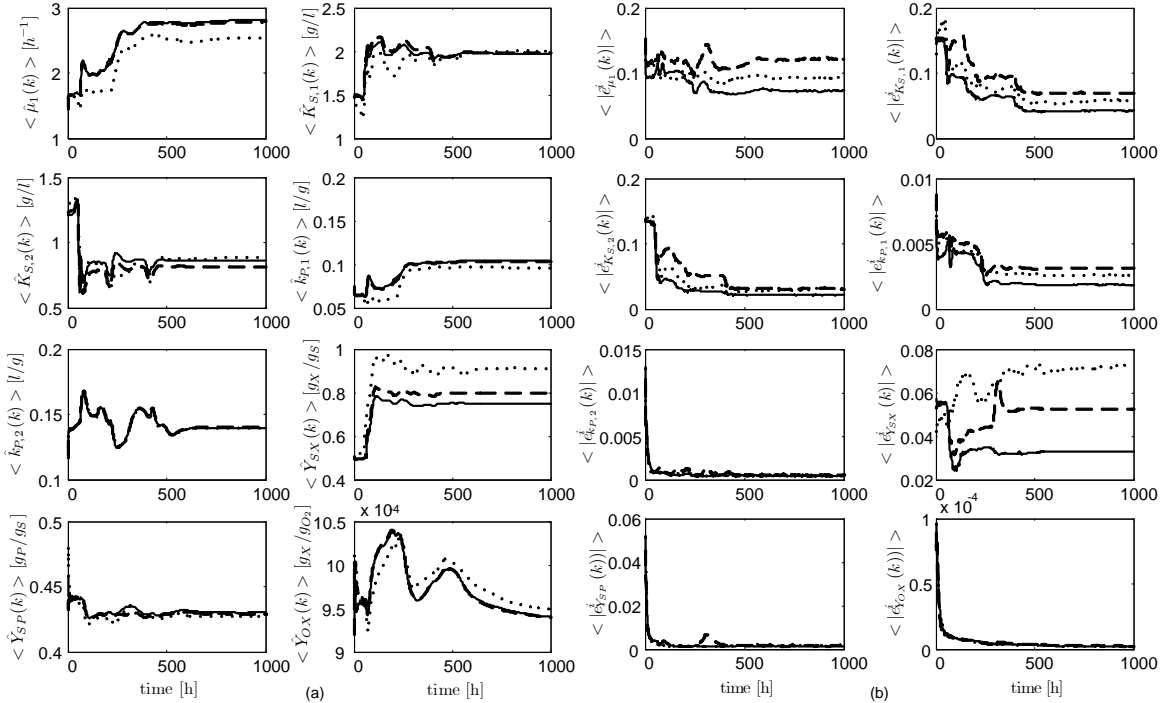


Fig. 8: *Intracellular* kinetic rate model parameter estimation results, averaged over 50 Monte Carlo simulations for the EKF (black solid line), the UKF (black dash line), and the EnKF (black dotted line): (a) mean parameter estimates $\langle \hat{\theta}_j^i(k) \rangle$ for every time index k; (b) mean absolute estimated parameter error, $\langle |e_{\theta_j^i}^i| \rangle = \langle |\hat{\theta}_j^i - \hat{\theta}_j^i(k)| \rangle$ for every time index k.

ulated data sets over 50 Monte Carlo simulations. Four different reaction rate models are used by the estimators while the simulated data sets are generated assuming that the *original* reaction rate parameters have been estimated experimentally. When using the *original* reaction rate model in the estimator, the best parameter estimation is achieved by the EKF with slightly poorer performances for the UKF and the EnKF. The lower performance of the UKF can be explained by the lack of tuning of its parameters. For the estimator using the *elementary* reaction rate model, the best parameter estimation corresponds to the UKF, while the EnKF has a poorer performance and the EKF diverges for some of the parameters. For the estimator with the *simplified* reaction rate model similar performances are achieved for the 3 estimators; the UKF slightly outperforms the other two. For the estimator with the *intracellular* reaction rate model, the best parameter estimation performance corresponds to the EKF.

The EnKF has a poor parameter estimation performance for most of the cases but when considering the mean RMSE of the estimated states it outperforms the other estimators for three of the four cases (see Table 7).

The computational cost of the estimators increases considerably from the EKF to the EnKF because of the number of projections required for every estimator iteration. The fermentation model is run from a Dymosim executable file and this slows down the computational performance of the estimators (i.e. the computational time required for every estimator iteration) mainly because Dymosim uses a slow file input/output interface. Despite this practical disadvantage, nonlinear estimators can be evaluated with complex Modelica models in a simple way. Our future work will focus on the parameter identifiability of the complete model.

References

- [1] P. S. Agachi, Z. K. Cristea, and A. Imre-Lucaci, *Model Based Control. Case Studies in Process Engineering*. Weinheim: Wiley-VCH Verlag GmbH&Co., 2006.
- [2] B. Lie and J. I. Videla, "Continuous bioethanol production by fermentation," in *Green Energy with energy management and IT*, Stockholm, 2008.
- [3] P. M. Doran, *Bioprocess Engineering Principles*. San Diego: Academic Press, 1995.
- [4] J. L. Crassidis and J. L. Junkins, *Optimal estimation of dynamic systems*, ser. CRC applied mathematics and nonlinear science series. Chapman & Hall, 2000.
- [5] S. F. Schmidt, *Application of State-Space Methods to Navigation Problems*, c.t. leondes ed. Academic Press, New York, San Francisco, London, 1966, vol. 3, pp. 293–340.
- [6] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME—Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.
- [7] D. Simon, *Optimal State Estimation – Kalman, H_∞ , and Nonlinear Approaches*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2006.
- [8] S. J. Julier, J. K. Uhlmann, and H. F. Durrant-Whyte, "A new approach for filtering nonlinear systems," in *Proceedings of the 1995 American Control Conference*, Seattle, WA, 1995, pp. 1628–1632.
- [9] S. Julier and J. Uhlmann, "A general method for approximating nonlinear transformations of probability distributions," tech. rep., RRG, Dept. of Engineering Science, University of Oxford, Nov 1996, Tech. Rep., 1996.
- [10] —, "A new extension of the Kalman filter to nonlinear systems," in *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*, Orlando, FL, 1997.
- [11] S. J. Julier and J. K. Uhlmann, "Unscented filtering and nonlinear estimation (invited paper)," in *Proceedings of the IEEE*, vol. 92(3). IEEE Institute of Electrical and Electronics, 2004, pp. 401–422.
- [12] G. Evensen, "The ensemble kalman filter: Theoretical formulation and practical implementation," *Ocean Dynamics*, vol. 53, pp. 343–367, 2003.
- [13] S. Gillijns, O. B. Mendoza, J. Chandrasekar, B. L. R. D. Moor, D. S. Bernstein, and A. Ridley, "What is the ensemble kalman filter and how well does it work?" in *Proceedings of the 2006 American Control Conference*, Minneapolis, Minnesota, USA, June 2006.

Model-Based Optimizing Control and Estimation using Modelica Models

Lars Imsland Pål Kittilsen Tor Steinar Schei
Cybernetica AS
7038 Trondheim, Norway
{lars.imsland,pal.kittilsen,tor.s.schei}@cybernetica.no

Abstract

This paper reports on experiences from case studies in using Modelica/Dymola models interfaced to control and optimization software, as process models in real time process control applications. Possible applications of the integrated models are in state- and parameter estimation and nonlinear model predictive control. It was found that this approach is clearly possible, providing many advantages over modeling in low-level programming languages. However, some effort is required in making the Modelica models accessible to NMPC software.

Keywords: Nonlinear Model Predictive Control, On-line optimization, process control, offshore oil and gas production

1 Introduction

Model Predictive Control (MPC) has become *the* advanced control strategy in the process industries [11]. MPC refers to control strategies which optimize future performance as predicted by a process model, and implement the first part of the calculated control inputs. The optimization/implementation is repeated at regular intervals to achieve robustness through feedback. Although *linear* MPC (based on linear, typically empirical, process models) is prevalent, it is seen that in many cases, MPC based on nonlinear process models (NMPC), with models derived from first principles and process knowledge, is advantageous or even necessary to achieve better control performance over varying operating conditions (due, for example, to varying product specifications or large process disturbances). In addition to the use of nonlinear process models, another important aspect with NMPC based on models from first principles, is that nonlinear state estimation is an essential part of the control system. NMPC has received considerable attention in academia, especially in terms of optimization methods

[1] and requirements for stability of the resulting closed loop [7]. However, when it comes to industrial application, use of NMPC clearly has an unfulfilled potential, although some applications are being reported, especially in polymerization processes [8, 11].

One important reason for the limited practical use of NMPC, is the substantial time and effort required for developing, validating and maintaining nonlinear process models that are valid over a wide operating range. Importantly, but sometimes overlooked, these models should at the same time be suitable for optimization, in terms of issues such as complexity and smoothness. An important step towards less costly model development is the use of advanced modeling environments, which promotes model structure, model reuse and model maintenance through equation-oriented modeling languages, object orientation and hierarchical composition of sub-models.

Literature reveals some effort towards using advanced process modeling environments in a practical dynamical optimization setting, e.g. [9], where gPROMS are connected to a software environment for dynamic optimization. However, the impression remains that this is very much a developing area.

The use of such models is not limited to NMPC in real-time process control settings. One can envision many types of real-time model-based applications using such models, ranging from data reconciliation, estimation (states, parameters, disturbances, soft-sensing) for monitoring and control, to advisory operator support systems and finally to NMPC. One can argue that a complete NMPC installation involves the other applications mentioned, such that if Modelica models can be used for NMPC, the other applications follows naturally.

The aim of this paper is to discuss requirements, challenges, opportunities, and experiences from using an advanced modeling environment, in particular Dymola/Modelica, for developing models that are used in model-based process control applications.

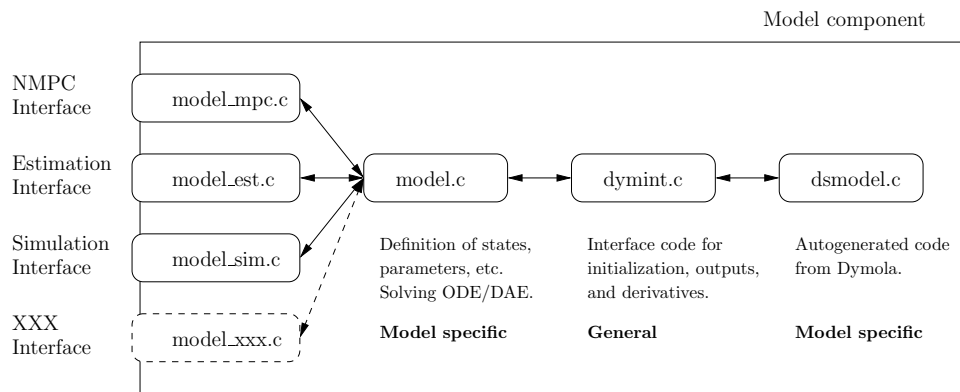


Figure 1: Illustration of model component software structure

The paper is structured as follows: First, we give some remarks on how we have integrated Modelica models (developed using the Dymola tool) into the NMPC tool CYBERNETICA CENIT (for state estimation and optimization). Next, we give some comments on modeling and model types, and give a brief overview over the modeling used in the case examples. In Section 4 we discuss using the Modelica models for state estimation, and give briefly some results obtained using real process data. In Section 5 we discuss optimization in Nonlinear Model Predictive Control, and illustrates with results from a simulation study.

2 Interfacing Modelica/Dymola models with NMPC estimation and optimization software

In this section, we discuss the integration of Modelica models with CYBERNETICA CENIT, a software package for NMPC developed by Cybernetica.

The CYBERNETICA CENIT kernel consists of three components: The NMPC optimization component, the state estimation component, and the model component. The components communicate (with each other and externally) using prespecified interfaces. The two first components are general, while the model component of course is specific for each project. Other CYBERNETICA CENIT modules, for example for offline parameter estimation/optimization for fitting the model to data, also exist and make use of the kernel, but are not considered part of the kernel.

The model component includes discretization (simulation of the model between sample intervals), such that the model is discrete time as seen from the state estimation and NMPC module.

Traditionally, the model component has been coded in C. This has served the purposes well, but for a number of reasons it is desirable to have a more user-friendly

way of implementing models, using a high-level modeling language. The overall goal is to reduce the cost of modeling, which is a significant cost factor in a NMPC implementation project. Reasons for the cost reduction include

- Promote reuse of models, also through building of model libraries.
- Better overview of models, ease of implementation and modifications.
- Easier exploitation of modeling effort in other contexts.
- Possibly easier integration of external models (external libraries, customer models, thermodynamics, etc.).

After an investigation of the available alternatives, evaluated against a range of criteria including the issues in the list above, it was found that Modelica was an excellent possible choice for an alternative modeling language. Moreover, the software tool Dymola provided a good Modelica modeling environment, and the opportunity to integrate the models in other software, through the Dymola C-code export option.

With the C-code export, the Modelica model is available in a C-file, `dsmodel.c`, along with interface functions. Figure 1 illustrates how this C-file can be integrated to form a model component ready to use with CYBERNETICA CENIT.

A distinct advantage of the C-code export offered by Dymola, is that it allows compilation of the total control system including model on any target system equipped with an ANSI C compiler. This is in contrast to systems which base the interface on software component interfaces such as CORBA, and requires (a version of) the modeling environment to run simultaneously.

On the other hand, it might be conceived as a disadvantage that the interface is Dymola specific, and not based on any standard.

Presently, the developed interface only allows obtain-

ing sensitivity information by finite differences, but with the availability of analytical Jacobian from Dymola models, a natural next step, subject for current development, is to include integration of sensitivities in the model component. This can be an advantage both for simulation of the model (see next section), but perhaps more importantly, also for solving the NMPC optimization problem (see Section 5).

3 Modeling and simulation

As mentioned in the previous section, the models developed in Modelica/Dymola must be solved/simulated in the model component. In this section we first give some general remarks on modeling and simulation for NMPC, and thereafter we briefly present the modeling that is done for the case examples in Section 4 and 5.

3.1 Simulating the model

Using equation-based modeling environments such as those based on Modelica, one generally ends up with differential-algebraic equation systems (DAEs). In Dymola, there are implemented algorithms for reformulation (symbolic transformation) of the DAE system such that it from the outside looks like an ODE system,

$$\begin{aligned}\dot{x}(t) &= f(t, x(t), u(t), p), \\ y(t) &= g(t, x(t), u(t), p),\end{aligned}\quad (1)$$

but where the evaluation of the right hand side in general requires the solution of some nonlinear equation systems. The reformulation ensures that these equation systems are as small, and hence as efficiently solved, as possible. However, the solution is based on iterative, local methods, such that it can in general take many iterations to find an acceptable solution, and worse, one is not always guaranteed to find a solution at all. (Although for well-behaved models, one normally finds a solution in few iterations.)

Another issue is that the right hand side might be discontinuous in its arguments. If this is the case, the solvers used to solve the (apparent) ODE above, must be able to handle discontinuities. Moreover, the system will often be stiff, calling for implicit methods with variable step lengths.

Apart from any possible discontinuities, the above issues (DAEs, stiffness, variable step lengths) do not in principle imply any problems using Modelica/Dymola models with an NMPC tool like CYBERNETICA CENIT.

Nevertheless, efficiency and robustness issues may change the picture. Simulation in a NMPC system

involves frequent resetting of system parameters (initial states, inputs and estimated parameters), which for DAEs in general requires online re-solving of the nonlinear equation set. For the ODEs exported by Dymola, it leads to frequent re-solving of the 'hidden' nonlinear equation sets.

If we can ensure that the model is a 'real' ODE (without nonlinear equation sets), this is avoided, resulting in increased speed and robustness.

There are no direct help in Dymola to avoid the nonlinear equation sets leading to a DAE system, but the reporting when translating models helps to identify where these nonlinear equations are.

Additionally, ensuring that the model is continuous, means that we can use more efficient solvers that do not have to handle discontinuities.

These issues require more effort during the modeling, and also imply that one often cannot apply other (library, customer) models directly. Nevertheless, the issues are important: In our experience, it is a key aspect of a successful implementation of a NMPC system to find the correct balance between computational complexity of the model/simulation and required model accuracy. Required model accuracy is not easily defined in general, but relates to the specific control objectives of the particular process. In this respect, more complex models are not necessarily more accurate.

When building models from physics, one typically ends up with stiff equation systems, which require implicit solvers with variable step sizes to be solved efficiently. If one chooses to exploit analytical Jacobians in connection with optimization (see also Section 5), this can in principle also be used in the implicit solvers to speed up computation.

3.2 Control-relevant modeling of an offshore oil and gas processing plant

In the North Sea (and on other continental shelves), oil and gas are produced by drilling wells into the ocean bed. From the wells, a stream of typically oil, gas and water arrive at a surface production facility which main task is to separate the components and make oil and gas ready for export, either through pipelines or by ship. A schematic picture of such an offshore oil and gas processing plant placed on an offshore platform is given in Figure 2. In this case, we have to some extent disregarded water, to concentrate on the oil and gas streams.

In Figure 2, we see that oil and gas enter from two different main sources (each main source is represented by one oil and one gas source) into three separators (the grey ovals). The separators are large tanks which split oil, water and gas. The produced oil is leaving in

the lower right corner of the figure, while the gas enters a compression train from the second and third separator. The compressor train, consisting of five compressors (five stages) compresses the natural gas for re-injection or to export through a pipeline (upper right corner). At the top, there is an additional gas import (from another production platform) with an additional gas compressor. Some of the gas is taken out (top left) as fuel for on board generators.

As can be seen, this is a fairly complex system in terms of numbers of components, however, many of the components are of the same type (mainly separators, compressors, valves, PID controllers, in addition to minor components such as sources, sink, splitter, sensors, etc.), which simplifies overall modeling.

A brief description of some unit models is given below:

- **Separators:** Separators are large tanks which due to their construction, and the different densities of the components, separate water, oil and gas into different process streams. The dynamics of the separator model is based on a mass balance and flash calculations to calculate the split of oil and gas. Based on the separator geometry (and thermodynamics), water and oil levels and gas pressure can be calculated from the component masses.
- **Compressors:** The centrifugal compressor models are static models based on compressor maps (specified by the compressor vendor) of polytropic head vs. volumetric rate, parameterized in compressor speed. The compressor maps are interpolated to yield continuous relations. The compressors are strongly nonlinear, that is, the gain from compressor speed (input) to pressure and volumetric rate are strongly dependent on operating point.
- **Valves:** There are different valve models for liquid and gas flow, both based on basic valve equations. Critical and sub-critical flow are handled. The valve characteristics can be chosen to be either linear or equal percentage via a drop-down menu.

For real-time efficiency reasons, we have made an effort to ensure that we end up with an ODE model. The main manifestation of this, is that we cannot have more than one unit that determines flow between each volume in the model. Therefore, we have introduced semi-physical 'nodes' (the grey round units in Figure 2), and tuned the volumes of these to retain good transient response (for example, by tuning them to be faster than the sample frequency, the exact value is not important in terms of simulation accuracy vs. measure-

ments).

Thermodynamics are important in order to calculate phase transitions between oil and gas. It is also essential to be able to describe the gas' properties over a large span in pressure. Furthermore, the model should have real time capabilities, favoring simple/explicit relations.

For phase equilibrium calculations, correlations of k -values (as function of temperature, pressure and molecular weight) were used together with a simplified representation of the many chemical species found in the real process. Gas density was described by a second-order virial equation, where the model coefficients were fitted to an SRK-equation for the relevant gas composition evaluated for the temperature and pressure range of current interest.

The thermodynamic models have been implemented in the style of the Modelica.Media library in the Modelica Standard Library.

4 State estimation

4.1 State estimation background

Nonlinear state-, disturbance- and parameter estimation are essential for NMPC implementations, but are also important in other settings than purely control-related, such as monitoring and surveillance, and static optimization/RTOs.

Estimation based on Kalman filter algorithms has become tremendously widespread over the last almost 50 years. Other types of estimation algorithms also exist, but are much less used. For nonlinear state estimation, Extended Kalman Filtering (EKF) algorithms should be used. Traditionally, these are based on analytical linearizations, but over the last years, it is seen that using divided differences (or similarly, Unscented Kalman Filtering (UKF) approaches) in many cases provides better performance than linearization-based EKF.

Importantly, the perturbation schemes used in connection with covariance update by divided difference-approaches (including UKFs) obtain information beyond linearization. Thus, for these cases, availability of analytical Jacobians from the model is not necessarily an advantage (unless it speeds up simulation). On the other hand, for estimation schemes based on linearizations (e.g. traditional EKF), or estimation based on numerical optimization (e.g. Moving Horizon Estimation (MHE)-approaches, taking inequality constraints into consideration), analytical Jacobians can be exploited.

CYBERNETICA GENIT has implemented EKFs based

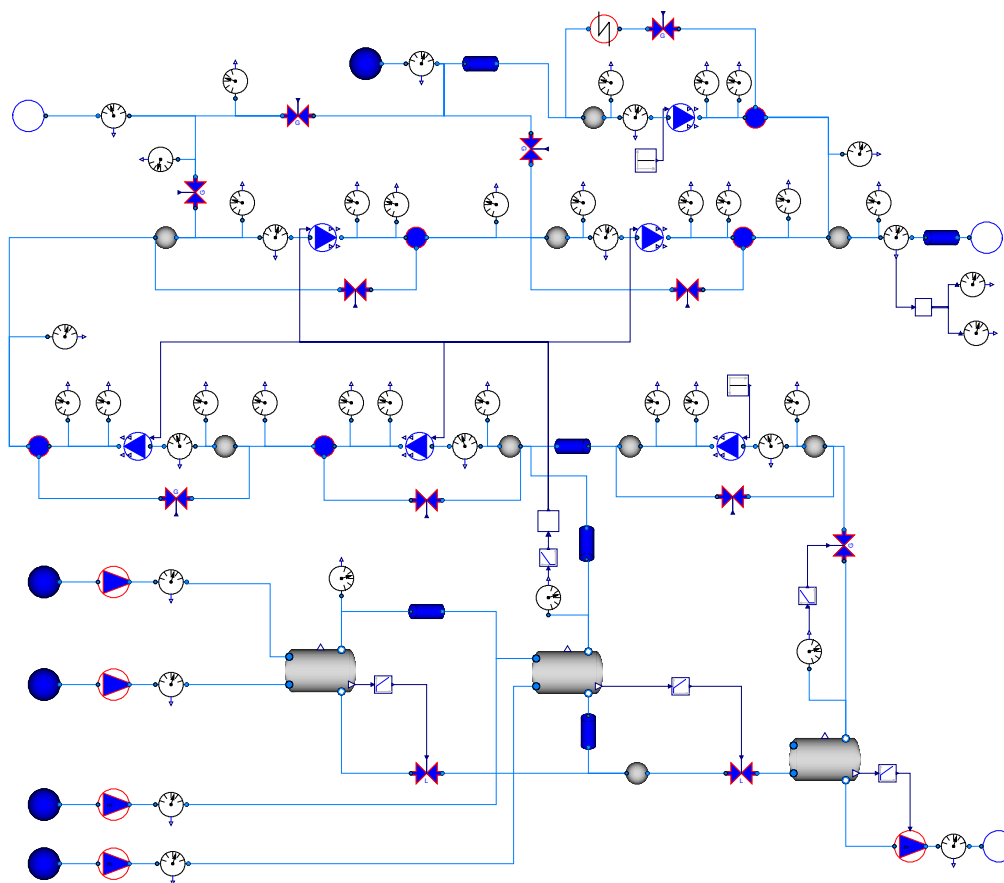


Figure 2: Overview of an offshore oil and gas processing plant, as implemented in Dymola.

on divided differences (both DD1 [13] and DD2 [10], in the notation of [10]), in addition to MHE [12]. For further information and discussion, see also [14].

4.2 Example: State estimation of offshore processing plant

The state and parameter estimation capabilities of CYBERNETICA CENIT (extended Kalman filtering based on finite differences in this case) was used to estimate states and model parameters in a Modelica model of the offshore oil and gas processing plant illustrated in Figure 2. The Modelica model was integrated as a CYBERNETICA CENIT model component as explained in Section 2. Logged data from real operation was used in the test.

The process is fairly well instrumented (a subset of the instrumentation is included in the Modelica model, see Figure 2), but there is no overall reconciliation of the individual measurements nor any overall measurement of key figures. From the individual measurements, most often in engineering units, it is hard to get an overview of the state of the process. With a complete process overview by the help of the model, it is possible to identify the current process state, being an es-

sential basis for taking the correct corrective actions in case of abnormal incidents, and also essential as a starting point for optimization of process operation.

The resulting ODE model of the system was fairly stiff, with modes ranging from around 0.1 seconds to hours, while the sampling time of the process was 1 minute. Therefore, it was absolutely necessary to use an (implicit) ODE solver with varying step lengths. In this case, the CVODE ODE solver¹ was used, with Jacobians found by finite differences. For this model, with 38 states and 35 estimated parameters, the state estimation ran more than 10 times faster than real time.

The state and parameter estimation was successfully tuned and tested on data from several days of operation. An excerpt is shown in Figure 3, where the model initially is simulated 'open loop', and the state estimation is turned on after 60 minutes. The figure demonstrates, for a single compressor stage, how the compressor parameters converge such that the estimated variables match the measured ones.

¹From the SUNDIALS package, see <http://www.llnl.gov/CASC/sundials/>.

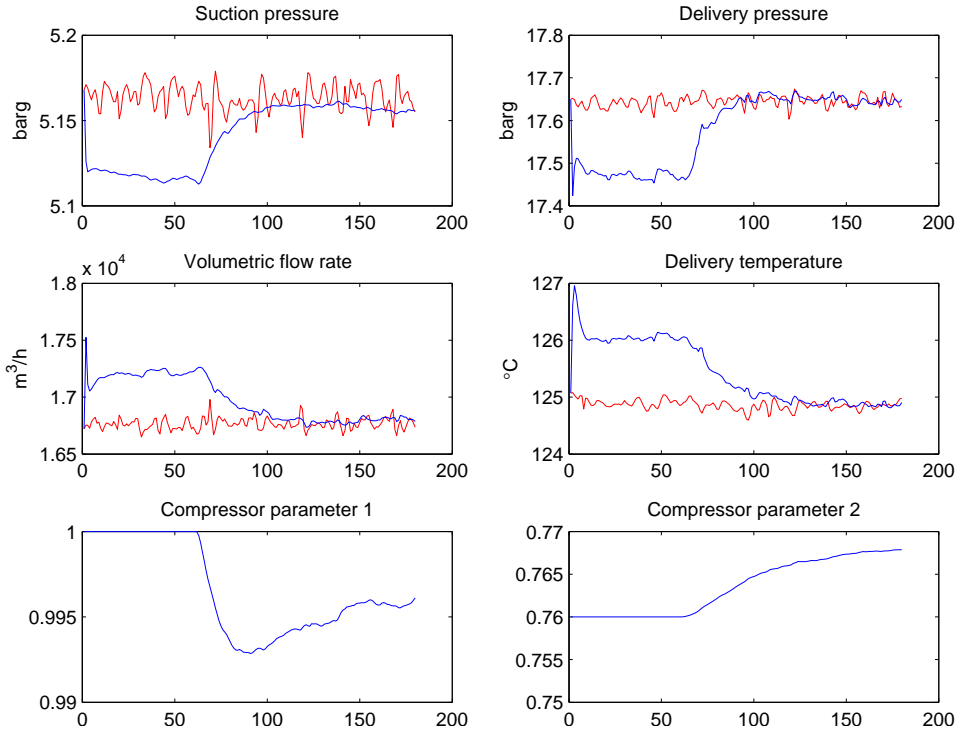


Figure 3: State and parameter estimation of one of the compressor stages. Red lines are real process data, and blue lines are estimated results.

5 Nonlinear Model Predictive Control

5.1 NMPC background

The NMPC optimization problem is a dynamic optimization problem, usually discretized to have a finite number of optimization variables (manipulated variables), that must be solved at regular (sampling) instants. The first part of the optimal solution – the first sample interval – is implemented to the process, before the dynamic optimization problem is resolved before the next sample instant. The optimization problem is using updated process information from a state estimation algorithm.

The optimization problem to be solved at time t , with available state estimate $\hat{x}(t)$, may look something like this, after a piecewise constant parameterization of future manipulated variables (u) over an horizon L :

$$\begin{aligned} \min_{u_0, u_1, \dots, u_{L-1}} \sum_{k=0}^{L-1} F(x_{k+1}, u_k) \text{ subject to} \\ \begin{cases} x_{k+1} - f(x_k, u_k) &= 0, k = 0, \dots, L-1, \\ x_0 &= \hat{x}(t), \\ h_x(x_k) &\geq 0, k = 1, \dots, L, \\ h_u(u_k) &\geq 0, k = 0, \dots, L-1. \end{cases} \end{aligned} \quad (2)$$

The discrete-time system $x_{k+1} = f(x_k, u_k)$ is in gen-

eral obtained by simulation of an ODE (1) over the sample intervals. The functions h_x and h_u represent constraints on states (or controlled variables) and manipulated variables.

In most cases, the (discretized) dynamic optimization problem is solved using numerical algorithms based on sequential quadratic programming (SQP). A SQP method is an iterative method which at each iteration makes a quadratic approximation to the objective function and a linear approximation to the constraints, and solves a QP to find the search direction. Then a line-search is performed along this search direction to find the next iterate. General SQP solvers may be applied to NMPC optimization, but it is in general very advantageous to use tailor-made SQP algorithms for NMPC applications.

Although a very crucial step in SQP algorithms tailored for NMPC optimization is the linesearch, the main approaches found in the literature are usually categorized by the way they specify the QP for finding the search direction. Arguably, the most common method is the *sequential* approach [5], which at each iteration simulate the model using the current value of the optimization variables (u_0, u_1, \dots, u_{L-1}) to obtain the gradient of the objective function (and possibly the Hessian), thus effectively removing the model equality constraints and the states x_1, x_2, \dots, x_L as optimization variables. Thereafter a reduced space QP prob-

lem is solved to find the search direction. Conversely, in the *simultaneous* [4, 2] approach, the model is implemented as explicit equality constraints, meaning that the optimization variables are both u_0, u_1, \dots, u_{L-1} and x_1, x_2, \dots, x_L . The third approach, *multiple shooting* [3, 6], can be viewed as a combination of the two other approaches, where, loosely speaking, the control horizon is divided into some 'sub-horizons' which are solved in a sequential fashion, and equality constraints link the sub-horizons.

There is no general consensus as to which of the above methods is best – probably, it is problem dependent. Note that the two latter approaches allow closer cooperation between the ODE/DAE solvers and SQP optimization than is revealed by the formulation (2).

A sequential approach to dynamic optimization is implemented in CYBERNETICA CENIT. A central issue is how to obtain the necessary sensitivity information for solving the NMPC optimization problem. The two main routes are either by finite differences directly on the objective function, or by integrating ODE/DAE sensitivities along with the model, and calculate NMPC sensitivities based on this. For the latter case, one can exploit the possibility of using analytical Jacobians in Dymola.

Calculating the gradient by finite differences means that many (depending on number of optimization variables) simulations over the control horizon has to be done, which can be time-consuming. Calculating the gradient based on sensitivity integration has the potential to be significantly more efficient, at least for some problems.

A possible problem with forming the NMPC objective function gradient (and possibly Hessian) based on ODE/DAE sensitivities, is that the resulting gradient (and Hessian) is not necessarily a very good approximation to the NMPC objective function to be solved numerically. Consider the following argument: By using finite differences directly on the NMPC objective function (which includes solving the ODE/DAE), we obtain a direct approximation to the gradient of the "numerical" NMPC objective function, which is what we are minimizing numerically. However, by computing the gradient based on ODE/DAE sensitivities, discretization errors will make the computed gradient different from the gradient of the numerical objective function.

Such errors may be important since one for computational complexity reasons is likely to push the accuracy limits for the ODE/DAE solvers.

5.2 Simulation example: NMPC of offshore processing plant

The case used in this section is similar to the one used in the previous section, but is based on (another) production platform. In this case, the focus is on the separation, and the gas compression is not modeled. The process has five different streams of oil and gas, that are to be separated in four separators (a separator train). In contrast to Section 4, the water phase is now explicitly modeled in the separators. The model was tuned to fit data from the real process, but all results shown in this paper are based on simulations.

The process is controlled by level controllers for water and oil, and gas pressure controllers for each separator. This is a standard solution, which works well in many/normal cases. However, in some cases, disturbances in the inlet flows from the inlet pipelines/wells can cause problems for the control of the separators. The levels in the separators will vary, which may cause bad separation and may be detrimental for equipment downstream the separators, due to uneven flow out of the separator train. The purpose for this study is to see if NMPC with state and disturbance estimation, using the level controller setpoints as manipulated variables (MVs), can exploit the buffer capacity in the separators to smooth out the outlet flows of water and oil. The oil is in this particular case entering a distillation column, and the water is entering a glycol regenerator, for regeneration of glycol that is added in the process. Smoother inflow to these units may allow more regular/increased production of the overall process.

There are six manipulated variables: The setpoints for water and oil level controllers in the separators (two of the separators does not separate water, and hence does not have a water level controller). The controlled variables (CVs) are pressures, levels and valve openings for all separators, and rate of change of glycol concentration in one separator.

The resulting model, with 29 states, was not particularly stiff. Therefore, a simple forward Euler ODE solver was used. The NMPC system, including state and disturbance estimation based on finite differences, and NMPC optimization with gradients found by finite differences, ran considerably faster than real time, using a sample interval of 6 s.

Some simulation results with a disturbance, a time-limited increased flow in one of the inflowing pipelines, are shown in Figures 4–6. Figure 4 shows how the NMPC reduces the level controller setpoints in the inlet separator (resulting in increased outflow valve openings, see Figure 6), to let the increased inlet flow (detected by the state and disturbance estimation) be smoothed out over all the separators. Figure 5

demonstrates how the NMPC achieves smoother out-flow from the last separator, and that the glycol fraction in the water varies less.

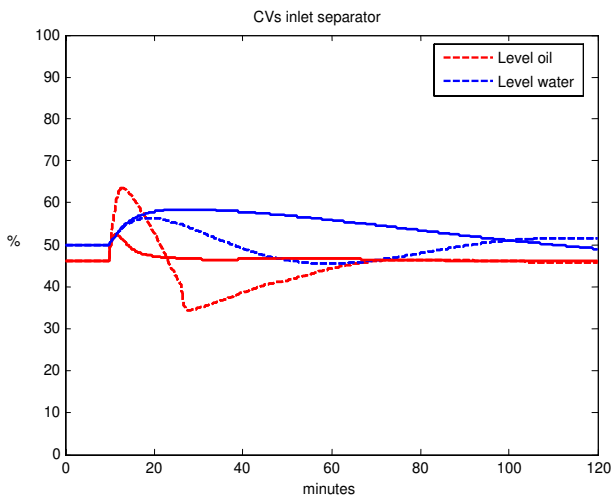


Figure 4: Oil and water levels in the inlet separator, with MPC (solid) and without MPC (dashed).

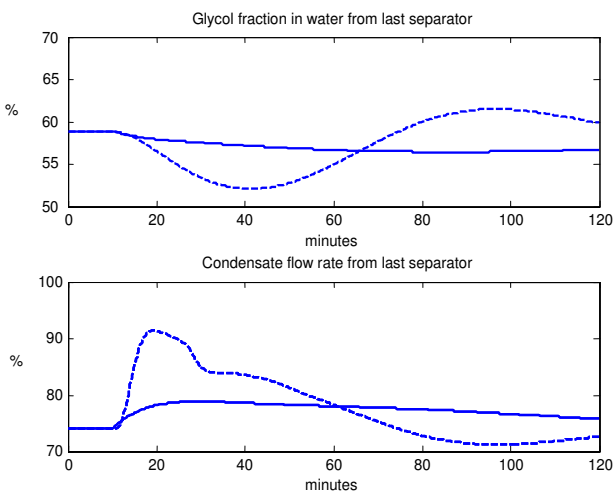


Figure 5: Glycol concentration in glycol/water mixture (top) and oil flow rate (bottom) from the last stage separator, with MPC (solid) and without MPC (dashed).

6 Experiences with using Modelica and Dymola for real time process control applications

In this section, we summarize some of our experiences with using Modelica and Dymola for process control applications.

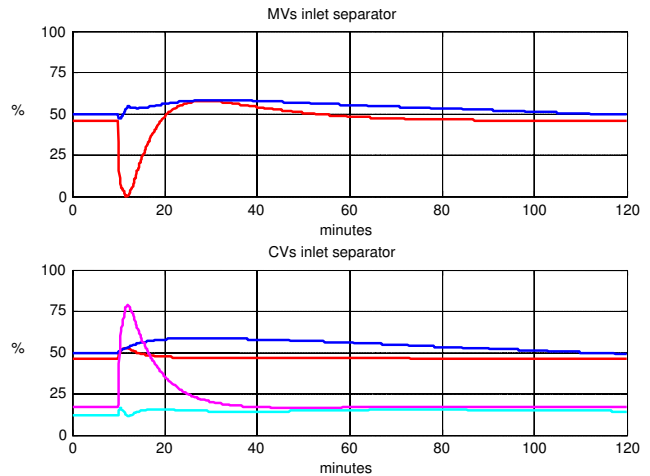


Figure 6: Level controller setpoints (MVs, top) and level and valve openings (CVs, bottom) in inlet separator. Red line is oil, blue line is water, magenta is oil valve opening, cyan is water valve opening.

6.1 Modelica modeling in Dymola

When it comes to modeling, Modelica and Dymola has much to offer over implementing the models in C. Due to the object orientation and the graphical interface it is easy to work on details and at the same time have an overview over the whole model. Using a tool such as Dymola, tasks like manipulation, testing and simulation of the model are convenient.

In this paper, we have used two cases from offshore oil and gas production. We saw some advantages in terms of reuse between these projects, but as we probably will work more in this area, we expect to see further advances at later stages. Using an object-oriented environment like Modelica, makes it easier to develop unit models with more general interfaces, such that they are easier reused. For some of the simple model units, we could use units from the Modelica Standard Library, although in most cases, some modifications were done. By drawing inspiration from Modelica.Media, we had a convenient structure for implementing the thermodynamics.

As with other equation-based modeling systems, debugging models during model development is a challenge in Dymola, and tools to help model debugging would be a benefit. However, by testing unit models thoroughly before aggregating them, many problems can be avoided.

When we have models with nonlinear equations systems (DAEs), we had in some cases problems with initialization of the equation systems, and identifying which variables that were part of the equation system. Of course, when making sure the model was an ODE, these problems were avoided.

6.2 Integration of Modelica/Dymola models in NMPC software

Using the C-code export option of Dymola, it was fairly straightforward to integrate the Modelica models as CYBERNETICA CENIT model components as described in Section 2. However, some modifications must be done to the Modelica model before it is exported, and some information, not directly available from the exported C-code interface functions, must currently be hand-coded into the model component. These issues are discussed below.

A significant part of the effort in constructing the model component based on the structure illustrated in Figure 1, is to generate and keep up to date the referencing/indexing variables in the file `model.c`. This information is necessary in the NMPC user interface, for instance for tuning of the EKF and the NMPC controller. This is presently coded by hand, but in theory, it should be possible to auto-generate at least large parts of this file based on the model information in the file `dsmodel.c`. Another possibility might be if Dymola added interface functions/functionality for this.

A NMPC system must exchange the following information dynamically with the model (in addition to states and state derivatives):

- Model inputs: The NMPC must (at least) be able to set the manipulated variables (MVs), the measured disturbances (DVs) and the parameters that are estimated by the EKF.
- Model outputs: Measured variables (for state estimation), and controlled variables (for NMPC).

We have (naturally) chosen to have the MVs and DVs as Modelica inputs. For estimated parameters, there is a choice involving some trade-offs:

- The estimated parameters could be part of the Modelica inputs. The advantages with this is that it is simple to manage in the model component, and that it is possible to calculate analytical Jacobians with respect to these parameters (for use for instance in state estimation using MHE, or in offline parameter estimation). The drawback is that the Modelica unit models must be modified to have these parameters as inputs, which makes it cumbersome to use the same model both for simulation and testing in Dymola, and as model for generating the model component.
- We can access the estimated parameters the same way as all other parameters². This makes 'book-keeping' of the parameters in the model component (`model.c`) more involved, and we cannot ex-

plot analytical Jacobians with respect to these parameters. On the other hand, this choice simplifies model maintenance, since we do not have to make new models for estimating parameters.

Presently, our implementation is based on the first choice, which in practice means we must maintain two Modelica models with identical behavior – one for simulation, and one for integration in the model component. This situation is not ideal. One possibility which might rectify the situation, is if Modelica had a variable type that is both parameter and input, and a kind of a 'master switch' that switches the interpretation.

In some cases, it would be an advantage to be able to debug the model code. Due to the structure of the auto-generated code, this is hard.

6.3 Running Modelica/Dymola models in NMPC software

There are some further interesting findings from the case study in Section 5.2. We had this model implemented as a model component in C before we implemented it in Modelica. By using profiling tools, we found that running NMPC with the model component based on the Modelica model, used less than 20% additional time compared to using the pure C model component, where most of the difference must be attributed to Modelica overhead since the models were practically mathematically identical.

However, to get the Modelica-based model to run this fast, we had to implement the Modelica functions used in the Modelica-model in C. Not surprisingly, there is considerable overhead in the implementation of Modelica functions, especially related to indexing of arrays. The possibility to implement Modelica functions in C is supported by the Modelica specification, and implemented in Dymola, and is a considerable practical advantage for real time applications.

7 Conclusions

It is possible to use Modelica/Dymola for modeling for NMPC purposes, with many of the advantages promised by such advanced modeling environments fulfilled. Such environments are helpful in developing complex process models, towards reuse of unit models, and we see potential for increased model value (by extending the application area of the model) and easier customer participation in model development.

However, using Modelica/Dymola models for NMPC has some hurdles. Some effort is required to make a Modelica simulation model ready to be used with

²Note that the estimated parameters will be constant in all simulations made, for instance over one sample interval in the EKF, or over the control horizon in the NMPC.

NMPC software. In our experience, two main issues are a) making Modelica model parameters accessible to the NMPC through Modelica inputs and outputs, and b) specifying the structure of state-, input-, output- and parameter vectors (e.g. for NMPC and state estimation tuning).

Finally, we emphasize that process models for NMPC should be developed with the specific task in mind, in terms of issues such as complexity, accuracy and smoothness. In some cases, this means that the model should be an ODE, while models from component-based modeling languages such as Modelica naturally translates into DAEs. It will in general require some effort and compromises for Modelica models to translate into ODEs.

Acknowledgments

We gratefully acknowledge StatoilHydro Research Centre in Porsgrunn, Norway for providing the cases and process data used in this paper.

References

- [1] L. Biegler. Efficient solution of dynamic optimization and NMPC problems. In F. Allgöwer and A. Zheng, editors, *Nonlinear Predictive Control*, pages 219–245. Birkhauser, Basel, 2000.
- [2] L. T. Biegler, A. M. Cervantes, and A. Wächter. Advances in simultaneous strategies for dynamic process optimization. *Chem. Eng. Sci.*, 57:575–593, 2002.
- [3] H. G. Bock, M. Diehl, D. B. Leineweber, and J. P. Schlöder. A direct multiple shooting method for real-time optimization of nonlinear DAE processes. In F. Allgöwer and A. Zheng, editors, *Nonlinear Predictive Control*, volume 26 of *Progress in Systems Theory*, pages 246–267, Basel, 2000. Birkhäuser.
- [4] A. M. Cervantes and L. T. Biegler. Large-scale dae optimization using simultaneous nonlinear programming formulations. *AIChE J.*, 44:1038, 1998.
- [5] N. M. C. de Oliveira and L. T. Biegler. An extension of newton-type algorithms for nonlinear process control. *Automatica*, 31:281–286, 1995.
- [6] M. Diehl, H. G. Bock, and J. P. Schlöder. A real-time iteration scheme for nonlinear optimization in optimal feedback control. *SIAM J. Contr. Optim.*, 43(5):1714–1736, 2005.
- [7] R. Findeisen, L. Imsland, F. Allgöwer, and B. A. Foss. State and output feedback nonlinear model predictive control: An overview. *European J. of Control*, 9(2-3):190–206, 2003.
- [8] B. A. Foss and T. S. Schei. Putting nonlinear model predictive control into use. In *Assessment and Future Directions Nonlinear Model Predictive Control*, LNCIS 358, pages 407–417. Springer Verlag, 2007.
- [9] Y. D. Lang and L. T. Biegler. A software environment for simultaneous dynamic optimization. *Computers & Chemical Engineering*, 31(8):931–942, 2007.
- [10] N. K. Poulsen M. Nørgaard and O. Ravn. New developments in state estimation for nonlinear systems. *Automatica*, 36(11):1627–1638, 2000.
- [11] S. J. Qin and T. A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003.
- [12] C. V. Rao and J. B. Rawlings. Constrained process monitoring: Moving-horizon approach. *AIChE J.*, 48(1):97–109, 2002.
- [13] T. S. Schei. A finite-difference method for linearization in nonlinear estimation algorithms. *Automatica*, 33(11):2053–2058, 1997.
- [14] T. S. Schei. On-line estimation for process control and optimization applications. In *Proc. 8th International IFAC Symposium on Dynamics and Control of Process Systems (DYCOPS-07)*, Cancún, Mexico, 2007.
- [15] T. S. Schei, P. Singstad, and Aa. J. Thunem. Transient simulations of gas-oil-water separation plants. *MIC—Model. Identif. Control*, 12(1):27–46, 1991.

Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models

Francesco Casella, Filippo Donida
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32 - 20133 Milano ITALY
e-mail: casella@elet.polimi.it

Bernhard Bachmann
Fachbereich Mathematik und Technik
Fachhochschule Bielefeld
Am Stadtholz 24 – 33609 Bielefeld GERMANY

Peter Aronsson
MathCore Engineering AB,
Teknikringen 1B, SE-583 30 Linköping, SWEDEN

Abstract

The formulation of steady-state initialization problems for fluid systems is a non-trivial task. If steady-state equations are specified at the component level, the corresponding system of initial equations at the system level might be overdetermined, if index reduction eliminates some states. On the other hand, steady-state equations are not sufficient to uniquely identify one equilibrium state in the case of closed systems, so additional equations are required. The paper shows how these problems might be solved in an elegant way by formulating overdetermined initialization problems, which have more equations than unknowns and a unique solution, then solving them using a least-squares minimization algorithm. The concept is tested on a representative test case using the OpenModelica compiler.

1 Introduction

The Modelica language is finding more and more applications in the field of thermo-fluid system modeling, due to the many advantages of the declarative, object-oriented approach. In this context, it is very often the case that steady-state initialization is required.

Specifying a well-posed steady state initialization problem in an object-oriented language is a non-trivial task for some fundamental reasons. From an end-user point of view, the ideal situation is to select a “steady-state initialization” option on the system components, without worrying too much about the actual internal implementation. This means that each component model should contain an initial equation section, with conditionally activated initial equations that express the steady state condition for that model. In this way, initial equations are specified locally within each model. Unfortunately, a well-posed initialization problem can only be formalized at the aggregate system level, i. e., on the system of DAEs describing the complete system. On one hand, index reduction can lead to a reduced number of states, if ideal pipes with zero pressure loss are used or ideal controllers are employed, so that some of the locally specified initial conditions are redundant. On the other hand, some model structures (e. g., closed systems) may be such that the locally specified steady-state conditions are not sufficient to completely determine the initial state.

The actual type and number of independent initial equations required to uniquely determine a consistent steady-state initialization thus depends in a non-trivial way on the connection topology of the system. It is therefore impossible for the library designer to write local steady-state initial equations

which are always good, because that depends on how the specific model will be connected to other ones. Furthermore, it is exceedingly hard for the end user to determine the exact structure of the required initial equations, because this would require a deep knowledge of the inner mathematical details of the single models, and of the mathematical properties arising from the interconnection of the models. The former requirement is against the principle of encapsulation: one should not necessarily be aware of the implementation details of an object in order to use it; the latter can be even more difficult for large systems.

The aim of this paper is to demonstrate how an elegant and user-friendly solution to this problem can be obtained by formulating overdetermined initialization problems, with particular reference to fluid systems. No extension to the Modelica language is needed. Three representative examples will be presented, then solved using the Open Modelica compiler and the methods presented in [1].

2 A Simple Circuit Model

The approach proposed in this paper will be demonstrated on a small case study: the simplified model of a heating circuit. The system includes an accumulator to pressurize the circuit, a pump, a heater (pipe with prescribed heat flow), a valve and a radiator (pipe with convective heat transfer to a fixed temperature sink), connected in a closed loop configuration (Fig. 1).

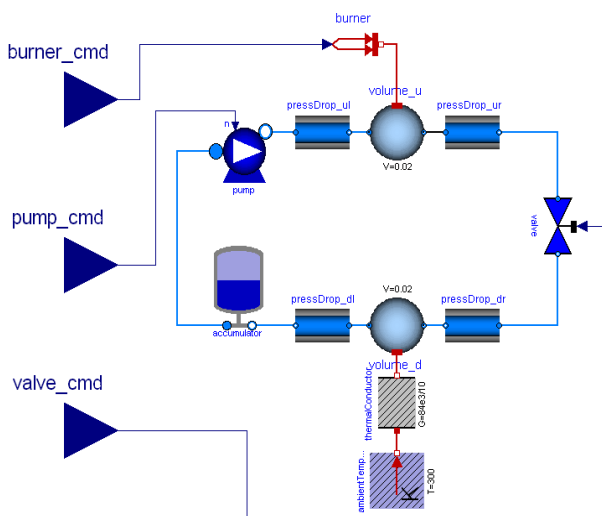


Figure 1. Flow diagram of the test case

The original model was built using components from the Modelica_Fluid library [2]. In order to overcome the current limitations of the OpenModelica compiler, the SimpleFluid library has been developed. The aim of this small library is to capture the essential mathematical structure of fluid system models, while avoiding advanced language features, such as the semiLinear operator and the replaceable packages of the Modelica.Media library, currently not supported by the compiler. These simpler models are more than adequate to demonstrate the proposed approach; the library will be updated with more complex models and test cases as the OpenModelica compiler is improved.

2.1 Connectors

The fluid connectors of the library are similar to the connectors of the ThermoPower library [3]-[4]:

```
connector FlangeA "Type-A connector"
  Types.Pressure p "Pressure";
  flow SI.MassFlowRate w "Mass flowrate";
  output Types.SpecificEnthalpy hAB
    "Specific enthalpy of fluid flowing A->B";
  input Types.SpecificEnthalpy hBA
    "Specific enthalpy of fluid flowing B->A";
end FlangeA;
```

```
connector FlangeB "Type-B connector"
  Types.Pressure p "Pressure";
  flow SI.MassFlowRate w "Mass flowrate";
  input Types.SpecificEnthalpy hAB
    "Specific enthalpy of fluid flowing A->B";
  output Types.SpecificEnthalpy hBA
    "Specific enthalpy of fluid flowing B->A";
end FlangeB;
```

Locally re-defined types are used in order to set reasonable non-zero default start values for the thermodynamic properties. The reader is referred to [3] for details about the connector design.

Thermal transfer is described by standard Modelica.Thermal.HeatTransfer connectors and components.

2.2 Medium models

Medium properties are computed by a replaceable medium model, similar to the BaseProperties model of the Modelica.Media standard library. The base model contains the pressure p , temperature T , density ρ , specific enthalpy h , and specific energy u of the fluid, as well as the partial derivatives with respect to pressure and enthalpy which are needed for the mass and energy balance equations.

The test cases described in this paper use a model of a compressible liquid with constant specific heat at

constant pressure, constant compressibility and constant thermal expansion coefficient.

2.3 Pump

Currently, a trivial pump model is employed, with prescribed flow rate: this could represent a pump equipped with an ideal mass flow rate controller. The prescribed flow rate is given by an input signal connector. The enthalpy increase due to the specific work added to the fluid is not taken into account, as it is negligible compared to the heat transfer in the heater and radiator models.

2.4 Accumulator

Accumulators are usually employed to pressurize liquid-filled circuit and accommodate the expansion and contraction of the fluid due to the thermal expansion effect. Typical accumulators are built using a tank partially filled with air, so that the amount of water contained depends on the air pressure. The model includes the three-way T junction to the circuit, so that it has two fluid connectors.

A simple linear model has been used to compute the amount of liquid contained in the accumulator:

$$M = Cp ; \quad (1)$$

where C is the hydraulic capacitance; since the pressure has been selected as a state, the mass balance equation of the model

$$\frac{dM}{dt} = w_1 - w_2 , \quad (2)$$

is written as

$$C \frac{dp}{dt} = w_1 - w_2 , \quad (3)$$

w_1 and w_2 being the inlet and outlet mass flow rates. Since the flow rate of fluid going into and out of the accumulator is usually much smaller than the flow rate in the circuit, trivial energy balance equations are assumed, where the specific enthalpy of the fluid going out of the T junction is always equal to the enthalpy of the incoming fluid.

The steady-state equation for this component, which contains a dynamic mass balance, should be

$$\frac{dM}{dt} = 0 ; \quad (4)$$

given the choice of states, the initial equation in the model is written as:

$$\frac{dp}{dt} = 0 . \quad (5)$$

2.5 Lumped volume

Mass and energy storage are represented by the classical lumped-parameter mass and energy balance equations. Pressure and temperature are used as states.

$$M = \rho V \quad (6)$$

$$U = M u \quad (7)$$

$$\frac{dM}{dt} = V \left[\left(\frac{d\rho}{dp} \right)_T \frac{dp}{dt} + \left(\frac{d\rho}{dT} \right)_p \frac{dT}{dt} \right] \quad (8)$$

$$\frac{dh}{dt} = \left(\frac{dh}{dp} \right)_T \frac{dp}{dt} + c_p \frac{dT}{dt} \quad (9)$$

$$\frac{dU}{dt} = M \frac{dh}{dt} + \frac{dM}{dt} h - V \frac{dp}{dt} \quad (10)$$

The mass and energy balance equations

$$\frac{dM}{dt} = w_1 - w_2 \quad (11)$$

$$\frac{dU}{dt} = w_1 h_1 - w_2 h_2 + Q \quad (12)$$

are thus written using the results of equations (8)-(10). Here, M and U are the mass and energy of the fluid contained in the model, V is the volume, c_p is the specific heat at constant pressure, h_1 and h_2 are the specific enthalpies of the fluid entering and exiting the volume, and Q is the heat flow entering the volume.

The steady-state equations for this component are:

$$\frac{dM}{dt} = 0 \quad (13)$$

$$\frac{dU}{dt} = 0 . \quad (14)$$

Given the choice of states, these equations can be more conveniently reformulated as

$$\frac{dp}{dt} = 0 \quad (15)$$

$$\frac{dT}{dt} = 0 . \quad (16)$$

2.6 Pressure loss model

In order to avoid trouble with hard nonlinearities at this stage, a simple linear pressure loss has been assumed:

$$w = K \Delta p , \quad (17)$$

where Δp is the pressure drop across the component, w is the mass flow rate through it, and K is a constant flow coefficient. Future version of the model will consider a density-dependent, quadratic pressure loss.

The energy balance is an isenthalpic transformation between the inlet and the outlet.

2.7 Valve

The valve model is similar to the pressure loss, except that the flow coefficient can be modulated by varying the valve opening input signal u from 0 to 1.

$$w = K u \Delta p . \quad (18)$$

2.8 Pipe

Each pipe is described by a simple symmetric lumped-parameter model, with one volume describing mass and energy storage, and two adjacent pressure loss models describing the momentum balance.

2.9 Choice of physical parameters

The nominal operating point of the circuit assumes a flow rate of 1 kg/s, a thermal power of 84 kW, and a convective heat transfer to the environment such that the temperature of the radiator is 10 K above the ambient value of 300 K, while the heater temperature is 330 K. The pressure loss in the valve is 1 bar, as well as the pressure loss in the pipes, which is equally divided between the two half-pressure-loss models. The hydraulic capacitance of the accumulator is 3 kg/bar.

3 Initialization problems

3.1 Steady-state initialization of a closed circuit

The components of the circuit model have 5 potential state variables: the pressure and temperature of the two volumes, and the pressure of the accumulator.

Since the circuit is closed, the total mass of the fluid in the circuit must be constant, because there is no mass flow rate entering or leaving the system. Therefore, the system equations, by their very nature, imply that

$$\sum_j \frac{dM_j}{dt} = 0 \quad (19)$$

where M_j are the masses of the fluid in the components with storage, i.e., the accumulator and the two pipe volumes. If one now sums the initial equations (4) and (13) for the accumulator and volume components, the same equation is obtained. This means that the simulation equations and the steady-state equations for a closed system will always be linearly dependent. The corresponding initialization problem is therefore singular, and has an infinite number of solutions, corresponding to different amounts of liquid in the circuit or, equivalently, to different levels of pressure in the circuit.

It is important to note that no single component has singular initialization equations: the singularity only arises at the system level. It is therefore convenient to leave all the steady-state equations in the single components, and add one more initial equation at the system level, e. g. by specifying the pressure at one point of the circuit or, alternatively, by specifying the total mass of liquid in the circuit. This leads to an overdetermined system of initial conditions, which has one more equation than unknowns, but now has one unique solution.

3.2 Steady-state initialization with zero-pressure-loss flow components

Suppose that the pressure loss due to friction in the radiator is small, compared to other pressure losses in the circuit. In order to avoid highly nonlinear stiff equations, and to reduce the number of states in the system, a possible modelling option is to neglect the pressure loss entirely, i. e. use the equation:

$$\Delta p = 0 \quad (20)$$

in place of equation (17) for the two pressure loss models of the radiator. This might be an interesting option for control-oriented models, where a reduced number of states is often sought.

As a consequence, the pressure within the radiator volume and the pressure within the accumulator are bound to be equal, so that the resulting system has index 2. The index reduction algorithm gets rid of one of the two pressure states, so that there now is one more redundant initial equation, compared to the previous case, even though the overdetermined system of equation still has one unique solution.

As in the previous case, this situation does not depend on equations which are local to a single sub-model, but rather depends on the system-level structure of the overall model, due to the way the sub-model are connected. It is therefore very convenient if the user doesn't have to change the local initialization option for any sub-model, and still get

the unique steady-state solution for the initialization problem.

3.3 Steady-state initialization with idealized controllers (inverse simulation)

So far, open-loop simulation problems have been considered, in which the three inputs corresponding to the three actuators (pump speed, valve opening, burner power) are prescribed functions of time. One could then study a closed-loop control problem, in which, e. g., the burner power is used to control the radiator temperature to a given set point, using a PI controller. In this case, one more steady-state equation would be needed for the controller state, but this would not cause any further imbalance between the initial states and the initial equations. However, it would be necessary to tune the parameters of the PI controller in order to obtain a stable and satisfactory performance.

In some cases, one could be interested in evaluating the transient of the control variable (the burner power) corresponding to some external disturbance, assuming a very tight control, without worrying about the actual tuning of the controller itself. This kind of study is carried out easily in an a-causal context, by just removing the equation which assigns the prescribed value to the control variable, and adding an equation which prescribes the value of the controlled variable to be equal to the set point. This kind of approach is also known as inverse simulation problem (see, e.g., [5]). The prescribed set-point must be smooth enough in order for the inverse simulation problem to have a well-defined solution, but this is outside the scope of the initialization problem.

In the specific case considered in this paper, one could prescribe the value of the radiator temperature, in order to obtain the corresponding value of the heater power input. This can be done both with the system described in Sect. 3.1, as well as with the system described in Sect. 3.2. In both cases, since the radiator temperature is one of the system states, the connection of the plant model to the idealized controller will enforce an algebraic constraint on a differentiated variable; index reduction will have to be applied in order to get an index-1 DAE, and thus one more state will be eliminated. Once again, since this is a system-level issue, it would be nice not to be obliged to change the initialization options inside any specific sub-model of the plant, but rather keep the resulting overdetermined initial equation system, which still has one unique solution.

4 Numerical results

The three test cases described in Sect. 3 have been set up in the SimpleFluid library, described in Sect. 2. The problems have then been solved using the OpenModelica Compiler (OMC) version 1.4.3 [6]. The current solution algorithm is summarized here:

- The Modelica code is flattened, obtaining the declarations of all variables, parameters and constants, as well as the full set of equations and initial equations.
- Index reduction is applied, in order to obtain a reduced-order, index-1 system.
- The initialization problem $f(z) = 0$ is built, by adding the initial equations to the set of index-1 DAEs of the system; z is the vector including the algebraic variables, the state variables, and the state derivatives, while f is the vector of the residual functions. Note that, in general, $\dim(f) \geq \dim(z)$.
- The initialization problem is then solved by minimizing the norm of the residual vector $F(z) = \sum_j f_j^2(z)$, by using the Sequential Quadratic Programming optimization code described in [7]; the start values of all variables are used as an initial guess for the iterative algorithm. If the initialization problem has one solution, the minimum is unique and characterized by a zero residual.

OMC successfully solves all the three initialization problems described in Sect. 3, finding the corresponding initial steady state, provided that:

- all the thermodynamic variables (pressures, temperature, densities) are given a meaningful, non-zero start value – this is accomplished by extending the standard SI unit types with suitable default start attributes within SimpleFluid;
- the pressure and temperature states of the volumes and of the accumulator are given a start value close enough to the steady-state value.

Unfortunately, convergence of the initialization problem seems to be rather sensitive to the start values of the temperatures in the volumes: a start value of 300 K instead of 330 K for the heater volume is enough to make the algorithm fail.

5 Improvements and future work

Several improvement actions are proposed in this section, which will be tested in future versions of OMC.

First of all, the size of the optimization problem corresponding to the initialization problem can be roughly halved by just removing alias variables from the flattened model. Although this sounds like a trivial operation, care must be exercised in order to avoid getting rid of user-defined start values, which might have been applied to only one of the variables in the alias set. For this purpose, it might be useful to define a suitable priority indicator for start value modifiers, and select the alias variable with the highest priority start value in the set.

In order to further reduce the size of the optimization task, BLT partitioning of the initialization equation set could be performed, in order to split the original problem into smaller problems, to be solved sequentially. As the incidence matrix has more rows than columns, one has more degrees of freedom in selecting the row/column permutation than it is possible in the standard square problem. This is an open topic for further research. Tearing methods could also be very beneficial in this context.

Better scaling must be ensured to improve the robustness of the minimization algorithm. Currently, the state and algebraic variables z of the initialization problem, and the equation residuals $f(z)$ are directly used in the optimization problem. Some equations and some variables thus have a predominating influence on the optimization problem, due to bad scaling. For example, in the test case discussed in this paper, the mass flow rates have an order of magnitude of 1, while the pressures are around 10^6 ; the mass balance equations have residuals (i.e., flow rates) of the order of 1, while the energy balance equations might easily give residuals (i.e. powers) of the order of 10^5 . This might explain the failure of the initialization algorithm even for small changes in the start values of the temperatures, since they mainly affect the energy balances, which have a larger influence on the residual norm than the mass balances.

To improve this situation, the algebraic and state variables might be normalized with their nominal values; the state derivatives might be normalized with the nominal values of the corresponding states, assuming a typical time scale of 1 second. On the equation side, residuals could be normalized with scale factors obtained by a Monte Carlo approach: these could be estimated by computing the residuals

with random small variations of the corresponding values around their start values.

Convergence of the minimization algorithm might be improved by introducing penalty functions which are added to the objective function when the unknown variables gets out of their min-max interval. In fact, confidence intervals for the initial value are usually known, which are much narrower than min-max values during simulation – new `minStart` and `maxStart` attributes for Real types could be defined in Modelica, in order to specify the range during initialization.

Finally, homotopy methods might be considered in order to improve the robustness of the convergence for not too accurate choices of the start values.

In order to be able to evaluate the impact of all these actions, it is important to be able to monitor the progress of the iterative minimization algorithm, step by step. Improved diagnostic features (e.g., logging of iteration variable values) should then be implemented in OMC, which could also be useful for the diagnostics of the nonlinear solvers during simulation.

As the robustness of the initialization algorithm and the diagnostic capabilities are improved, it will be possible to increase the complexity of the test cases, first by introducing density-dependent, quadratic pressure losses in flow models, and then by trying more complex systems with larger numbers of equations and states.

6 Conclusions

Steady-state initialization problems for fluid systems are often naturally specified in terms of overdetermined systems of initial equations, having more equations than unknowns, but possessing just one unique solution. These problems can be solved using minimization algorithms. The paper motivates the need of such problems with reference to a simple test case, and presents results obtained with the OpenModelica compiler. Suggestions to improve the robustness of the OpenModelica solver are also given. The Modelica source code of all the test cases is available from the authors; contributions to improve the algorithms within the OpenModelica Compiler are welcome.

7 References

- [1] Bachmann B., Aronsson P., Fritzson P.. “Robust Initialization of Differential-Algebraic Equations”, *Proceedings of the 5th Modelica Conference*, Vienna, Austria, 4-5 Sep 2006, pp. 607-614. <http://www.modelica.org/events/modelica2006/Proceedings/sessions/Session6a2.pdf>
- [2] Casella F, Otter M., Proelss K., Richter C., Tummescheit H., “The Modelica Fluid and Media library for modeling of incompressible and compressible thermo-fluid pipe networks”, *Proceedings of the 5th Modelica Conference*, Vienna, Austria, 4-5 Sep 2006, pp. 631-640. <http://www.modelica.org/events/modelica2006/Proceedings/sessions/Session6b1.pdf>
- [3] F. Casella, A. Leva, “Modelling of thermo-hydraulic power generation processes using ModelicaModular Modelling in an Object Oriented Database”, *Mathematical and Computer Modelling of Dynamical Systems, Modelling of Systems*, v. 12, n. 1, pp 19-33, 2006.
- [4] F. Casella, A. Leva, “Modelica open library for power plant simulation: design and experimental validation”, *Proceedings 3rd International Modelica Conference*, Linköping, Sweden, Nov 2003, pp. 41-50. http://www.modelica.org/Conference2003/papers/h08_Leva.pdf
- [5] M. Thümmel, G. Looye, M. Kurze, M. Otter, J. Bals, “Nonlinear Inverse Models for Control”, *Proc. 5th International Modelica Conference*, Hamburg, Germany, Mar 2005, pp. 267-279. http://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf
- [6] Peter Fritzson, et al. “The Open Source Modelica Project”, *Proc. 2nd International Modelica Conference*, 18-19 March, 2002. Munich, Germany See also: <http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html>
- [7] M. J. D. Powell, “The NEWUOA software for unconstrained optimization without derivatives”, *Proc. 40th Workshop on Large Scale Nonlinear Optimization*, Erice, Italy, 2004, paper DAMTP 2004/NA05.

Session 3c

Automotive Applications

Modeling of Conventional Vehicle in Modelica

Wei Chen, Gang Qin, Lingyang Li, Yunqing Zhang, Liping Chen

CAD Center, Huazhong University of Science and Technology, China

chenw@hustcad.com

Abstract

Modelica is a modern language used to model physical systems. The language is object-oriented, non-causal and the models are mathematically described by differential algebraic equations. The characteristic of modelica language make it very suited to define model libraries with reusable components, model complex applications involving parts from several application domains, and many more useful facilities.

InteDrive library was created for simulating automotive driving performance, fuel consumption and emissions. The library is yet under developed by Huazhong University of Science and Technology in modelica language. The aim of the library is to provide the user with an easy to use and highly replaceable set of vehicle component models, and predict the vehicle performance, especially fuel consumption for a given cycle.

The main components of this library and their applications are introduced in this paper. The simulation was carried out by MWorks, which is a general modeling and simulation platform developed by Huazhong University of Science and Technology. The simulation results were compared with ADVISOR. The easy and fast modeling process shows that modelica is very useful for the modeling and simulation of vehicles.

Keywords: InteDrive library; Simulation; Modelica; MWorks

1 Introduction

Automotive manufacturers have been striving for decades to produce vehicles which satisfy customers' requirements at minimum cost. Many of their concerns are on fuel economy, road performance and driveability. Improving fuel economy is both a political concern of alleviating dependency on foreign fuel and a customer preference of reducing vehicle operating cost. Consumers also expect vehicles to provide satisfactory performance with desirable driving comfort. So it is very necessary to

predict the vehicle performance when the vehicle is design.

There have been a lot of program tools to predict vehicle performance, for example, cruise by AVL, ADVISOR, PSAT, etc. However, these tools are block-oriented and demand a huge amount of manual rewriting to get the equations into explicit form. Hence, these tools are less extensible, and hard to reuse. It brings too much inconvenience to the user.

Modelica is an object-oriented language for modeling of large and heterogeneous physical systems. The language is object-oriented, non-causal and the models are mathematically described by differential algebraic equations. These characteristics make it fast in modeling and easy to reuse modeling knowledge[1]. Modelica has been used to model various kinds of systems and proved to have superiority over traditional tools in modeling efficiency, especially over Matlab/Simulink.

InteDrive library was developed in modelica language to predict automotive driving performance, especially for fuel consumption. The aim of the library is to provide the user with an easy to use and highly replaceable set of vehicle component models.

MWorks is under developed by Huazhong University of Science and Technology. It is a general modeling and simulation platform for complex engineering systems which supports visual modeling, automatically translating and solving, as well as convenient postprocessing. The current version is based on Modelica 2.1 and implements almost all the syntax and semantics of Modelica.

A vehicle model was built with InteDrive library, and the simulation was carried out with MWorks. The simulation results are compared with ADVISOR, and show the correctness of the model.

2 Components

The InteDrive library contains some components of a conventional vehicle, such as engine, clutch, gearbox, etc. The modeling process have referred the paper[2-4]. The present structure of InteDrive can be viewed in Fig.1.

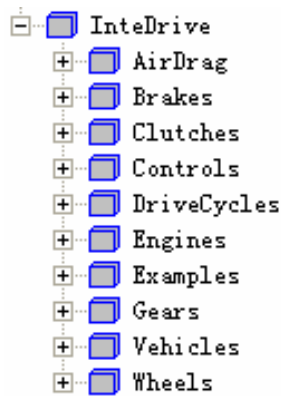


Fig.1 The structure of the InteDrive

The components of the InteDrive library are described as below.

2.1 Vehicles

The vehicle is one of the main objects in a model. This component contains general data of the vehicle, such as nominal dimensions and weights. The library presents only dynamic models for the longitudinal motion of the vehicle. So a sliding mass may represent as vehicle body. The model will be developed in the future for considering the load transfer to the rear or front axle when the vehicle is accelerating or braking.

2.2 DriveCycles

The drive cycle is the vehicle speed trace versus time. It is very useful for the evaluation of fuel economy and emissions. When the simulation was carried out, the vehicle speed must follow the speed profile to calculate the fuel consumption and emissions. This package includes tables for several driving cycles. At present, it contains the NEDC, UDDS, NYCC, HWFET and some standard driving cycles. Any other cycle can be added easily if desired.

Interpolation to the cycle table was used to get the vehicle speed when the simulation was carried out.

2.3 Clutches

The clutch contains the model of a friction clutch as used in cars with manual gear boxes. It is controlled by the driver via the clutch pedal position. In this paper, we adopt the clutch model in modelica standard library, and made some modification for simplification. The maximum normal force was changed to maximum transferable torque. So the parameters of the clutch may be acquired more easily.

2.4 Brakes

This is described by braking data and dimensions. By the implementation of a specific braking factor it is possible to model disc brakes as well as different forms of drum brakes. In this paper, a brake model in modelica standard library was adopted.

2.5 Gears

This package contains the gears in a vehicle, such as gear box, differential, final drive, etc.

The engine torque is turned into a power take-off torque by considering the transmission, the mass moments of inertia, the moment of loss. The modeling of gear box can be referred to the paper[3-5].

2.6 Engines

The component engine contains a model for a combustion engine. The engine was modeled by a structure of characteristic curves and maps. As the characteristic curves for the full load, the fuel consumption and others can be freely defined by the user. It is possible to define a gasoline engine as well as a diesel engine. Interpolation to the fuel map was used to get the fuel consumption. The emission can be calculated also if the emission map was defined.

2.7 Wheels

The wheels and tires link the vehicle to the road. In this paper, a block called IdealGearR2T in modelica standard library was used to model the wheel. It converts the rotational motion to translational motion. A force acted on the wheel to model the rolling resistance.

2.8 Controls

The vehicle is controlled to make the vehicle speed follow the driving cycle profiles. The controls include throttle control, brake control, gearbox control, clutch control and so on.

The throttle and the brake are controlled by PI controllers. The input to the PI controller was the error of vehicle speed acquired by simulation and the speed requested by the driving cycle. If the vehicle speed exceeds the reference speed, the driver controls the brake to let the vehicle slow down. If the vehicle speed is lower than the reference speed, the driver controls the throttle to let the vehicle accelerate. The PI parameters were tuned to control the vehicle properly.

The gearbox controller shifts the gears according to the vehicle speed. It is necessary to define the up- and downshifting velocities always only for one gear less than are available in the gear box (i.e. for a five step gear box, only for four gears the up- and

downshifting velocities have to be defined). As can be seen in figure 2, the upshifting velocity of the 2nd gear means that at this velocity the gear box control is upshifting from the 2nd into the 3rd gear. The downshifting velocity for the 2nd gear means that at this velocity the gear box control is downshifting from the 3rd into the 2nd gear.

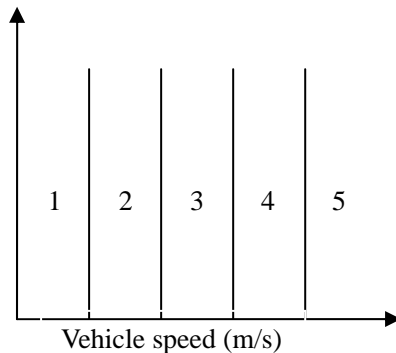


Fig. 2 Shift strategy by vehicle speed

The clutch control determines whether the clutch should be fully engaged, fully disengaged in this paper. The state of the clutch depends on the requirements of the drivetrain.

If the gear is changing, for an upshift or downshift, the clutch is disengaged. If no (positive) torque is required of the engine and or the speed required of the engine is less than its idle speed then the clutch is disengaged. Otherwise, the clutch is engaged.

2.9 AirDrags

This component models the air resistance force act on the vehicle. Usually, the aerodynamic resistance force is approximated by simplifying the vehicle to be a prismatic body with a frontal area A_f . The force caused by stagnation pressure is multiplied by aerodynamic drag coefficient c_d that models the actual flow conditions.

$$F_a = \frac{1}{2} \rho_a A_f c_d v^2$$

Here, v is the vehicle speed and ρ_a is the density of the air. The parameter c_d must be estimated using CFD programs or experiment in wind tunnels.

3 MWorks

MWorks is a general modeling and simulation platform for complex engineering systems which supports visual modeling, automatically translating and solving, as well as convenient postprocessing. It

is under constant developing by Huazhong University of Science and Technology. The current version is based on Modelica 2.1 and implements almost all the syntax and semantics of Modelica.

MWorks has features as follows:

- With modern integrated development environment styles, it provides friendly user interfaces such as syntax high-lighting, code assist etc.;
- Based on object-oriented compiler framework, it perfectly supports almost all the syntax and semantics of Modelica;

Using self-adapting solving strategies, it can agilely solve differential equations, algebraic equations and discrete equations.

MWorks Studio is a visual modeling environment which supports drag-drop modeling based on Modelica Standard Library. It is also an integrated development environment integrating with translator, optimizer, solver and postprocessor.

As a developing tool, this studio provides many modern IDE styles to promote the users' conveniences just as Eclipse or Microsoft Visual Studio does, such as real-time syntax highlighting, content assist, code formatting, outlining etc.

The snapshot of MWorks Studio is shown as Figure 3.

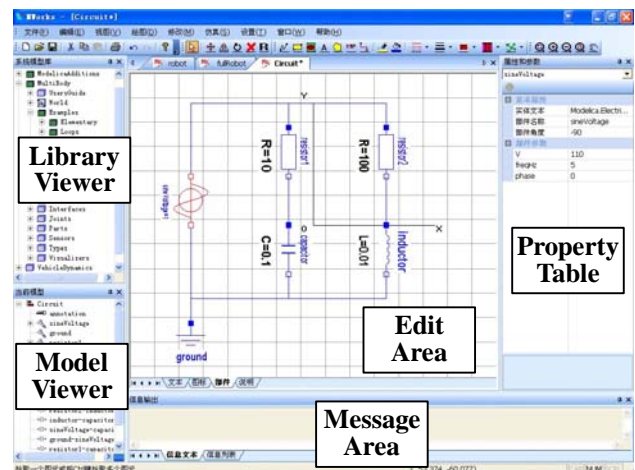


Fig. 3 Snapshot of MWorks Studio

The library viewer illustrates all predefined system libraries, all loaded user libraries and other top models in the memory. The model viewer shows all components of the current model. The edit area is visual modeling, text modeling, icon-editing or information area, and the status is chosen by tag. The property table displays all properties of selected element in the model, and the properties can be edited here. The message area displays all messages

in the checking, translating, or simulating, including status and error messages. The error can automatically be located by double clicking error message.

The auxiliary functions of real-time syntax highlighting, content assist, code formatting and outlining are provided in the text modeling status.

The solver of MWorks includes two primary modules: collection of algorithms and console of solving strategies. Solver provides different basic algorithm alternatives for users to select appropriate one. Now, a series of algorithms for different kinds of equations have been collected in the solver, such as SUNDIALS.

More details about MWorks can be referred to paper [6].

4 Simulation

A complete conventional vehicle was built by drag components from InteDrive library. The vehicle model can be seen in figure 4. The vehicle was modeled by a forward-facing approach include the driver model, which controls the throttle, brake, clutch, gearbox to make the vehicle speed follow a given driving cycle.

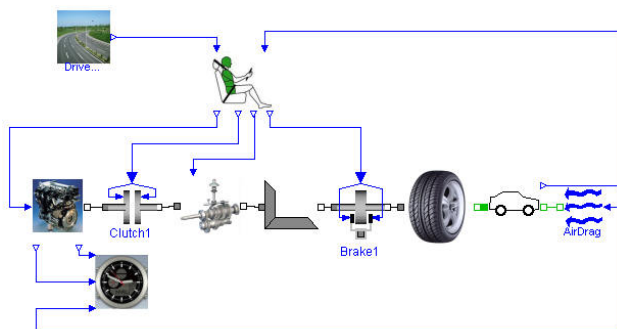


Fig. 4 Vehicle model

The key parameters for the vehicle simulation are listed in Table 1.

Table 1 Key parameters of the vehicle

Components	Key Parameters
Engine	Maximum power: 41kW@5700rpm Maximum torque: 81Nm@3477r/min
Final Drive	3.77
Transmission	3.5676/2.008/1.3289/1.0/0.7525
Vehicle Mass	1000Kg
Wheel Radius	0.282m
Rolling Resistance	0.09
Wheelbase	2.6m
Frontal Area	2m ²
Coefficient of Air Drag	0.335

The other parameters can be referred to ADVISOR2002, with the vehicle config file called CONVENTIONAL_default_in. The simulation works was carried out with MWorks.

The vehicle is driven with the UDDS driving cycle and the actual vehicle speed is given in figure 5. The total distance of the UDDS was 11.991 km. The difference between desired speed from the UDDS driving cycle and the actual speed is shown in figure 6. It seems the vehicle has been controlled as desired. The signal from the gear box can be seen in figure 7. The fuel information was shown in figure 8. The fuel use was 0.69 liter at the end of the cycle. So if the vehicle drives a distance with 100 kilometers, the fuel consumption was 100/11.991*0.69=5.75 liter. The result of ADVISOR is 5.9 liter. It shows the correctness of our models.

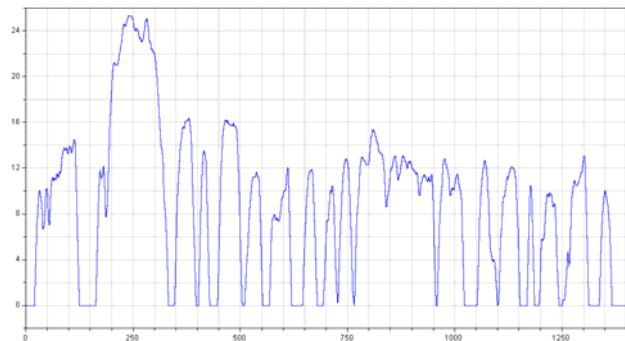


Fig. 5 Actual speed for the reference car during the UDDS driving cycle. X-axis shows time [s], y-axis vehicle velocity [m/s].

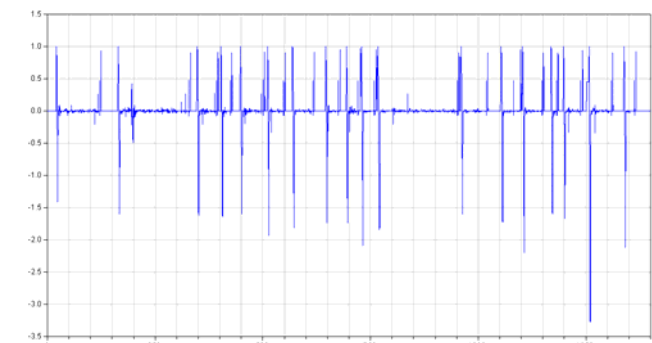


Fig.6 Difference between UDDS speed and actual vehicle speed during the simulation. X-axis shows time [s] and y-axis shows velocity [m/s].

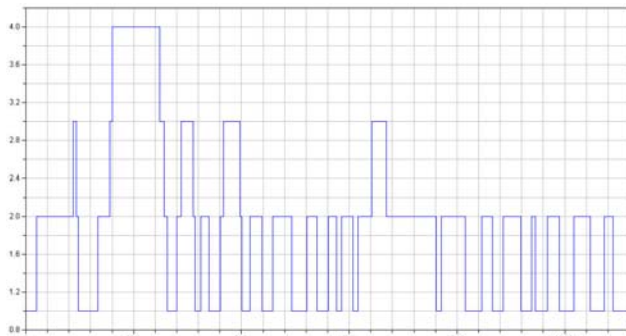


Fig.7 Gear number during the simulation. X-axis shows time [s] and y-axis shows gear number.

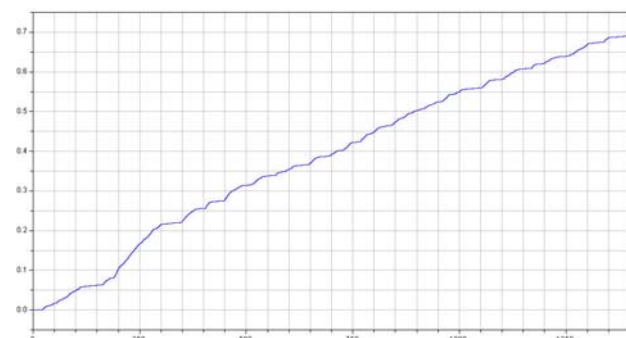


Fig.8 Fuel consumption during the simulation. X-axis shows time [s] and y-axis shows fuel [L].

5 Conclusions

In this paper, a library for modeling of simulation automotive fuel consumption is introduced. It uses the interfaces from the Modelica and ModelicaAdditions packages to be compatible with other libraries. The easy and fast modeling process shows the superiority of modelica language in modeling.

InteDrive provides various vehicle component models to simplify for the user to build the vehicle model according to their needs. The modular structure of the model design allows to take advantage of the Modelica language.

6 Future work

The library is under constant development. The models in the library will be modified with more detailed description. Much more models such as viscous-clutch, torque converter, CVT, and so on will be developed. Much more components with electrical modules will be developed, and the library is aimed to model conventional vehicles and electric vehicle.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (Grant No. 60574053), the National High-Tech Development 863 Program of China (Grant No. 2006AA110105).

References

- [1] Modelica Association. <http://www.modelica.org>.
- [2] Fritzson, P., et al. The Open Source Modelica Project. Proceedings of the 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany.
- [3] P. Nobrant. Driveline Modelling using MathModelica. Master's thesis. Linkopings universitet, Linkoping, Sweden, 2001.
- [4] Johanna Wallen. Modelling of Components for Conventional Car and Hybrid Electric Vehicle in Modelica. Master's thesis. Linkopings university, Linkoping, Sweden, 2004.
- [5] Otto Montell. Advanced concepts in Modelica and their implementation in VehProLib. Master's thesis. Linkopings university, Linkoping, Sweden, 2004.
- [6] FAN-LI Zhou, LI-PING Chen, etc. MWorks: a Modern IDE for Modeling and Simulation of Multidomain Physical Systems Based on Modelica. Proceedings of the 5th International Modelica Conference, 2006.

Vehicle Model for Limit Handling: Implementation and Validation

Johan Andreasson
Modelon AB
Ideon Science Park
SE-223 70 Lund, Sweden
E-mail: johan.andreasson@modelon.se

Mats Jonasson
Department of Vehicle Dynamics and Active Safety
Volvo Car Corporation
SE-405 31 Göteborg, Sweden
E-mail: mjonass2@volvocars.com

Abstract

This paper describes how a vehicle model from the VehicleDynamics Library is configured, parameterised and validated for predicting limit handling manoeuvres. Particular attention is paid to the selection of subsystem models with suitable levels of detail, as well as the selection of measurements performed and measuring equipment. A strong principle running throughout the presented work is component-based design where parameterisation is performed on subsystem levels, no tuning on the final vehicle models is done. As a final test, the vehicle model is exposed to a sinusoidal steering input. It turns out that the model is able to reproduce the vehicle's behaviour for the driving scenario selected up to the limit of adhesion.

Keywords: Vehicle Dynamics, Component-Based Modelling, Limit Handling, Validation

1 Introduction

Safety plays a prominent part in the development of vehicles. A large portion of the development is devoted to vehicle stability and the control task to maintain stability even under severe driving situations. Here, multi-body modelling becomes a powerful tool to preserve competitiveness and keep the development time within the given timescale. One reason for this strength is the ability to offer an improved understanding of the vehicle and also to support the ranking of its' design variables without any access to the physi-

cal vehicle [1]. Moreover, the development of vehicle control is facilitated if the vehicle plant pose an adequate response.

Driving conditions such as strong side motion of a vehicle are often considered to be unsafe, since the driver risks losing control of the vehicle. Such potentially dangerous situations need to be identified, and accordingly, there is a great deal of interest in reproducing this class of scenarios. However, the combination of fast transients and high accelerations triggers strong non-linear vehicle characteristics, which in turn make great demands on the model used. One example of this manoeuvre is the single lane change [2]. An example of such a *limit-handling* manoeuvre is given in Figure 1.

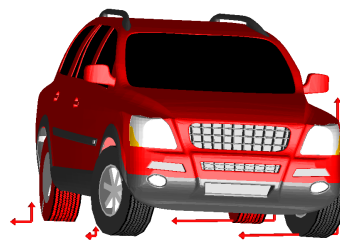


Figure 1: Simulation of the vehicle model undertaking a severe lane change maneuver. The arrows visualize the forces generated in the tyre contact patch.

A fundamental requirement when considering simulation as an alternative to real-life testing is validity. A model must not just be valid in the sense that it captures the results of already tested scenarios and pa-

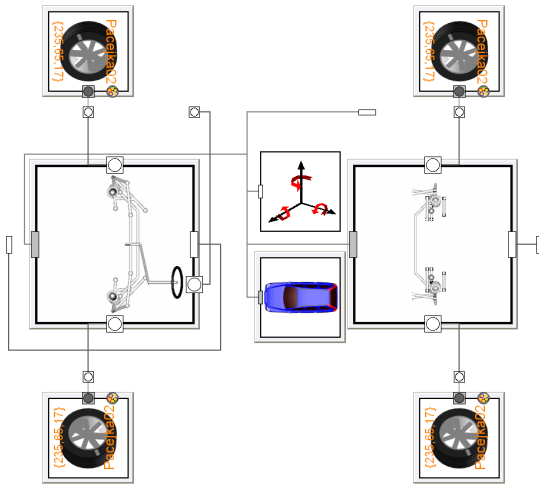


Figure 2: Diagram view of the chassis model layout with body, suspensions and wheels. The component above the body keeps track of the vehicle motion and handles initialisation.

parameterisations. It must especially be able to predict the effects of new scenarios, parameters and configurations, and therefore, the aim of this work is to show that by having valid subsystem and component models, the resulting vehicle model shall also be valid.

This paper will demonstrate how a vehicle model from the VehicleDynamics Library (VDL) [3] is put together from parameterised subsystems and verified against limit handling measurements. Special attention is given to choice and configuration of vehicle model, and parameterisation of subsystems, especially suspension characteristics.

2 Vehicle Model Configuration

The test vehicle is equipped with front McPherson and rear multi-link suspensions. The chassis model is implemented as a multi-body model in VDL with a rigid body onto which the suspensions and wheels are mounted as illustrated in Figure 2.

For the suspensions, there are two main approaches to modelling the kinematics and compliance characteristics, respectively. The kinematics can either be specified by the hard point locations of the links or as tabular characteristics depending on wheel travel (and steering for the front suspension). Compliance is either given by the individual characteristics of the elastic elements or as a lumped characteristics of the whole suspension. The required tabular characteristics and lumped elasticities can be generated from kinematics and compliance (K&C) analysis.

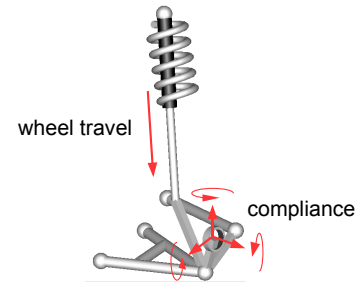


Figure 3: Animation view of the left front suspension linkage with the 7 DOF indicated.

In this application, suspension is modelled as ideal kinematic multi-body linkages with one degree-of-freedom (DOF) for wheel travel. The compliances that are caused by bushings and material deflection is lumped in one element between the wheel carrier and the hub. This approach is similar to what has been used in e.g. [4] and is illustrated in Figure 3.

The compliance adds 6 DOF for each suspension linkage, with wheel travel totally 7 DOF. Together with the front steering compliance there is a total of $7 \times 4 + 1$ DOF with 2 states each for position and velocity, i.e. 58 states for the suspensions.

The wheel models use the Pajeka'02 tyre force model in VDL, implemented according to [5]. This representation was chosen because it is considered to be state of the art and because there were available tyre data in this format. The tyre force model has two states for lateral and longitudinal relaxation lengths (first order dynamics). Together with the wheel's spin DOF, there are therefore 4 states per wheel and additionally, there are 6 degrees of freedom (12 states) for the vehicle's body motion, giving in total $58 + 4 \times 4 + 12 = 86$ states for the chassis model.

3 Suspension parameterisation and verification

As already mentioned, the model used contains both component parameters and lumped characteristics. Most parameters are taken from construction data such as geometries, masses and inertias, but some are calculated from measurements on isolated subsystems. Here, this is illustrated for the suspension compliance characteristics.

The kinematics and the compliance of the front and rear suspension have been measured in a dedicated rig where the car body is fixed and a post is mounted on

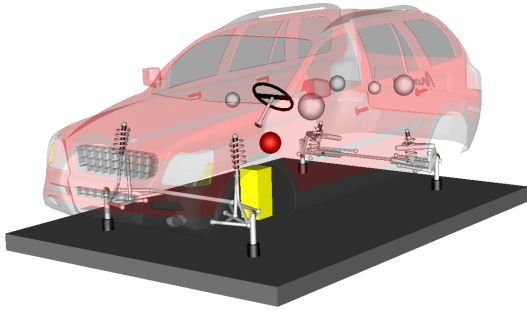


Figure 4: Virtual version of the test rig used for kinematics and compliance analysis. The chassis body is kept fixed and one actuator at each hub applies forces while the motion is registered using cameras. The spheres indicate centre of gravity and payloads.

each wheel hub (Figure 4). Using this post, forces and torques can be applied to replicate different driving scenarios. In this case, forces have been applied in the lateral (\hat{e}_y), longitudinal (\hat{e}_x), and vertical (\hat{e}_z) directions, both at the wheel's centre (C) and at the estimated tyre-road contact point (W). For every load case, the rotations (\bar{p}) and translations (\bar{r}) of the hub are measured.

For a force applied anywhere other than at the hub, there is both a resulting torque (\bar{t}) and a force (\bar{f}) at the hub so by comparing two equal forces applied at different locations, the torque dependency can be calculated, and thus

$$\underbrace{\begin{pmatrix} \bar{f} \\ \bar{t} \end{pmatrix}}_{\bar{F}} \rightarrow \underbrace{\begin{pmatrix} \bar{r} \\ \bar{p} \end{pmatrix}}_{\bar{\Delta}}. \quad (1)$$

Assuming that the dependency is linear, equation 1 can be rewritten as $\bar{\Delta} = \mathbf{C}\bar{F}$ where \mathbf{C} is a 6x6 compliance matrix and \mathbf{C}^{-1} the corresponding stiffness matrix, which is required for the compliance element.

As described in Section 2, there are 7 degrees of freedom for each suspension, 6 from lumped compliance element and 1 from wheel travel. Unfortunately, from a numerical point of view, it is hard to separate these dependencies since springs in a car are a factor >100 more compliant than the contribution from the compliance element. However, since the deflection in the compliance element is small in comparison with the total wheel travel, it is assumed that the accuracy requirement of the z-deflection from the compliance component is low. By keeping the vertical position of the measured hub fixed while forcing the opposite wheel hub to move, a force (f_z^l) is implied through the stabiliser linkage. This affected the measured hub ($\bar{\Delta}'$)

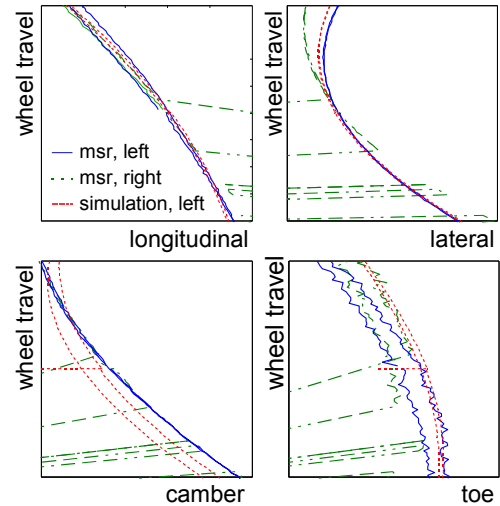


Figure 5: Suspension characteristics showing left and right side measurements (blue, green) and simulation (red). Longitudinal, lateral, camber and toe motions on the plot's x-axis and wheel travel on the y-axis.

so that a new column could be calculated from $\bar{\Delta}'/f_z^l$. This column is used as a replacement in C and the original column is used to define the spring rates. Correspondingly, the linear compliance of the steering system is also extracted from these measurements.

With \mathbf{C} calculated for each suspension linkage, the wheel travel tests performed in the K&C rig are carried out virtually with VDL to verify the behaviour of the suspension models. Figure 5 shows a comparison between the K&C measurement and the corresponding simulation for different wheel travel from.

Since it is a well-known fact that hydraulic dampers may deviate from the specification, all four dampers were disassembled and measured in a damper rig. The non-linear force-velocity characteristics retrieved from the damper measurements were used for purpose of modelling. This was carried out by linearising the characteristics piecewise for the compression and expansion phases respectively. One complicating aspect is the ability of the rear dampers to adapt to load changes. In brief, this can be explained as a preload in parallel with the damper that adapts slowly to the vehicle's load and the driving conditions.

4 Validation of the vehicle model

As already mentioned, limit handling involves both fast transients and highly non-linear characteristics. To create a validation in such circumstances, it is sufficient to measure the state-trajectory of the vehicle body. However, even if the response from the sim-

ulations coincides with the measured trajectory, little information can be retrieved about the correctness of the subsystem models used.

In order to identify and improve the modelling and parameterisation of these subsystems, it is advisable to extend the handling measurements to include even more mechanical phenomena. Obviously, since the tyre forces contribute substantially to the vehicle motion, added to which they are well-known to be hard to model, they become an important source to be monitored. In addition, at limit handling the vehicle executes large roll or/and pitch motions, and as a consequence, the suspension deflections become large. For large deflections, there is a significant alteration in toe and camber as illustrated in Figure 5, which in turn influences the tyre forces. Moreover, the compliance characteristics changes during the deflection. The most obvious situation is the entrance of the bump stop for large suspension compressions. For these particular reasons, the supervision of the deflection of all four corners becomes a viable option.

4.1 Instrumentation selection

Keeping the information discussed above in mind, the vehicle was equipped with a gyro-platform, four torque measuring wheels and sensors for deflection of all four corners. The gyro-platform measures the rotations (roll, pitch and yaw speeds) and accelerations (in x, y and z axes) of the car body. As illustrated in Figure 6d, the gyro-platform was mounted between the front seats. The standard wheels were replaced by the torque measuring wheels, which are able to measure tyre forces and wheel torques in and around x, y and z axes. This is possible due to strain gauges positioned at the rim. The suspension deflection instrumentation comprises levelling sensors, which measure the distance between the wheel hubs and car body.

In addition, signals from Controller Area Network (CAN) were logged in order to monitor wheel speeds and the states of the engine, brakes, gears and Haldex differential. All signals (approximately 90) were collected and sampled at 50 ms in a computer. Finally, a steering robot was mounted to support for steering input at a high level of accuracy and repeatability. Figure 6 illustrates the measurement setup of the vehicle. One important issue was to judge and assign a span for the accuracy of these measurements. Another challenge related to the large amount of redundant data retrieved from the vehicle instrumentation. One example of this redundant data relates to vehicle speed, which can be taken from wheel speed sensors (from CAN)

and also via the gyro-platform. To select the best data, information from sensors was compared, and later on, consolidated or arbitrated. In addition to this, the vehicle's corner weight and ride height were measured manually.

4.2 Driving scenarios

The driving scenarios were selected with two purposes in mind; reference and validation. To meet the first requirement, tests were carried out under conditions that allowed the measuring equipment to be tested as independently of the vehicle where possible. Typical examples are to expose the vehicle to constant conditions such as gradients in different directions.

Tests to measure the vehicle behaviour during steady-state manoeuvres were performed with both purposes in mind, and included, braking and acceleration to verify longitudinal load transfer and the resulting pitch of the body. Also steady-state cornering was executed by driving with a constant radius of 45 metres, while gradually increasing the vehicle speed up to the maximum achievable lateral acceleration. Thus body roll and tyre normal load distribution could be validated.

The handling manoeuvres used solely for validation, were selected to cover as much of the dynamics of the vehicle as possible up to its limits. This group of tests were conducted to force the vehicle into transient motions:

- Step steer using the steering robot
- Single lane change, sinusoidal steering input from the steering robot
- J-turn and simultaneous relief of the gas pedal (oversteer situation)
- Double lane change with a driver

Finally, the test procedure above was repeated under conditions where roll and yaw stability control were deactivated, in the test data presented in this report.

4.3 Validation results

Validations was finally achieved on the most extreme manoeuvres which is illustrated here by a single lane change test with all active safety systems turned off. The vehicle response, both measured and simulated, is shown in Figure 7. The amplitude and the vehicle speed are set to reach the limit of available grip to trigger most non-linear characteristics. From the slight bouncing in the roll angle it can be seen that the bump



Figure 6: The tested vehicle equipped with a) wheel travel sensors, b) steering robot, c) measuring wheels and d) gyro platform

stops in the suspensions are activated which also gives the same effect on lateral acceleration. It can also be seen that the side slip angles follow each other closely throughout just over half of the manoeuvre when they start to diverge. During the first half, the two simulations (red and green) are practically the same for all vehicle states while later on, the red and green are closer. At the time when the severe single-sine tests were carried out, the proving ground was slightly moist, as opposed to the rest of the testing period when the ground was dry. A qualified guess is that μ was slightly lower when these manoeuvres were performed, which is also supported by the fact that the simulation with μ gives a behaviour closer to that measured. Another interesting aspect is the fundamentally different results for $\mu = 1$ and $\mu = 0.95$. While the first simulation shows a vehicle that slowly recovers low side slip, the latter one continues to spin out and never recovers. During this type of severe maneuvers, even very slight changes to the surrounding conditions can have a great impact on handling behaviour.

Another effect that is of particular importance during manoeuvres where a bump stop is involved is the ride height. The sudden change in load transfer can give completely different results, as illustrated in Figure 8, showing the effect of a change in ride height. The red curve shows the lateral acceleration for the setup with the measured ride height and is the same as the red curve in Figure 7. The green curve shows the same setup, but with the default ride height taken from construction parameters. The default ride height was higher in the front and lower at the rear, compared to the measured ride height. As a result, the default settings gives a later bump stop activation and thereby slower turn-in and more roll motion.

The influence on the suspension elasticity on the vehi-

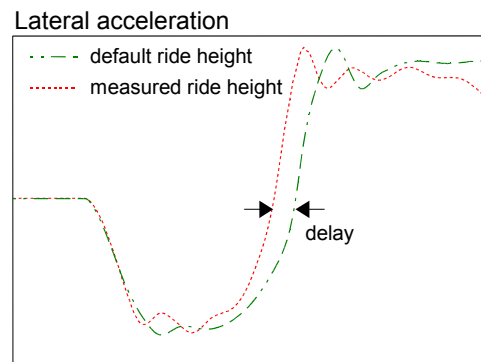


Figure 8: Lateral acceleration for two different ride height settings, the phase delay is 0.18s.

cle characteristics is well-known and provides an important means to tune a vehicle's characteristics. This effect is also seen in many of the manoeuvres investigated. However, for the limit manoeuvre presented in in Figure 7, changes in compliance have less effect on the results. As an example, a change of the rear compliance by a factor of 10 only gives very slight changes to the trajectory. This is expected to be due to the high amplitude which makes the front tyres reach their saturation limit almost immediately, thereby the load transfer has a significantly higher effect on the generated lateral force compared to changes in wheel angles [6].

5 Conclusions

This paper has presented a methodology for validation of a vehicle model, which is to be used in a broad range of driving scenarios. A suitable approach for the selection of wheel suspension parameters has been pre-

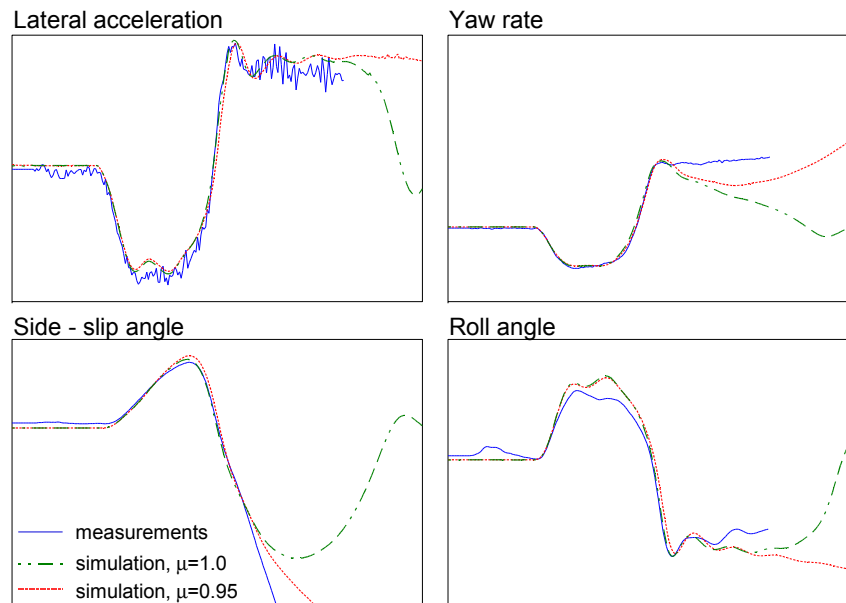


Figure 7: Response of open loop sine excitation (single lane change) at 80 km/h, measurements (blue) and simulations with $\mu = 1$ (green) and $\mu = 0.95$ (red). Lateral acceleration (upper left), yaw rate (upper right), side slip angle (lower left) and roll angle (lower right).

sented. In addition, the wheel suspension models have been validated through measurement in a chassis rig. A strong principle throughout the presented work is component-based design where parameterisations are done on sub-system levels and no tuning on the final vehicle models is performed.

The final validation involves reproduction of a real driving scenario, which has been represented here by measurement of the state-space trajectory. The results indicate that it is feasible to design a valid vehicle model, at least up to limit handling, from valid sub-systems without involving additional tuning. Finally, it is demonstrated that a minor error in the estimation of unknown environmental factors, such as road friction, risk to jeopardise the correspondence.

From these findings, it is evident that the methodology presented is a viable tool for use in the vehicle developments. In addition, it has a great potential to support for the development of safer vehicles and facilitate the development of preventative safety functions.

Acknowledgments

This work was financed in part by the Swedish National Energy Agency. The authors are grateful to Mr Per Hesselund and Mr Mikael Riikonen at Volvo Car Corporation for assisting us with the vehicle measurements. In addition, we would like to thank Dr Bengt

Jacobson at Volvo Car Corporation and Professor Annika Stensson Trigell at KTH Vehicle Dynamics for their immense support.

References

- [1] M. Blundell, D. Harty, *The Multibody Systems Approach to Vehicle Dynamics*, Elsevier Butterworth-Heinemann, 2004.
- [2] Technical Committee ISO/TC 22, Road vehicles - Transient open-loop response test method with one sinusoidal input, ISO/TR 8725, 1988.
- [3] J. Andreasson, M. Gävert, *The VehicleDynamics Library - Overview and Applications*. In: *Proceedings of the 5th Modelica Conference*, Vienna, Austria, Modelica Association, 4-5 September 2006.
- [4] K. Salani and G. Heydinger, *Parameter Determination and Vehicle Dynamics Modeling for the National Advanced Driving Simulator of the 2006 BMW 330i*, SAE paper 2007-01-0818.
- [5] Pacejka, H.B. *Tyre and Vehicle Dynamics*, Butterworth-Heinemann, 2002.
- [6] J. Andreasson, *On Generic Vehicle Motion Modelling and Control*, Ph.D. thesis, ISBN 91-7178-527-2, 2006.

Modelling of a Double Clutch Transmission with an Appropriate Controller for the Simulation of Shifting Processes

Henrik Isernhagen Clemens Gühmann
Technische Universität Berlin
Chair of Electronic Measurement and Diagnostic Technology
Sekt. EN 13, Einsteinufer 17, 10587 Berlin
{henrik.isernhagen, clemens.guehmann}@tu-berlin.de

Abstract

In this paper the modelling of a double clutch transmission with an appropriate controller is presented. An accordant library for modelling different levels of detail and the use of defined state signals are introduced. Furthermore, the control of the transmission with the simulation of shifting cycles is discussed. By varying the driver models it is possible to drive miscellaneous drive and shifting cycles. We present simulation results of a drive cycle with an examination of the interaction between the transmission control and the engine control. Finally, the application of the model and the simulation data are shown in view to the parameterisation of an automated measurement data analysis system.

Keywords: double clutch transmission, power train, control, shifting processes, state signals, simulation

1 Introduction

Nowadays, more and more new cars are assembled with double clutch transmission because of efficiency, drive comfort, and uninterrupted power shifts. A double clutch transmission contains two parallel transmission shafts with two parallel clutches. The clutches can be either dry clutches or laminar multi-disc (wet) clutches. The different gears are mounted alternately on the two transmission shafts. The first shaft contains the odd gears, and the second shaft contains the even gears. Depending on the number of gears the reverse gear is mounted either on the first shaft or on the second shaft. There are uninterrupted power shifts possible by reason of the two different clutches with the accordant shafts. The first clutch opens and the second clutch closes simultaneously during the shift process resulting in an uninterrupted power transmission. The correct control of the two clutches is very

important in view to the different shifting processes. At this point it is necessary to distinguish between upshifts and downshifts as well as pulling power and pushing power of the power train. Control and calibration errors can result in bad shifting, e. g., in form of revolution speed droppings, break outs, or oscillations. These errors should be detected by an automated measurement data analysis system [1]. The software is used for the evaluation of power train measurement data in the vehicle development. The parameterisation of this analysis system requires measurement data of good and bad shifting processes. Accordingly, the aim of the modelling and the simulation are to get data of shifting processes of different quality.

In this paper we present the modelling, control, and simulation of double clutch transmissions. First, a basic library and the use of state signals are introduced in Section 2. In Section 3 we discuss the need for models with different levels of detail. Section 4 shows the control structure of the double clutch transmission in view to different shifting processes, and Section 5 describes the simulation of drive and shifting cycles. Finally, we discuss in Section 6 simulation results of particular shifting cycles, and we present an outlook to the parameterisation of an automated measurement data analysis system [1] using the simulation results.

2 Library for modelling double clutch transmissions

For the modelling of double clutch transmissions we developed a library with basic components. On the one hand there are auxiliaries, interfaces, and basic blocks, which are reused in further blocks and models. These blocks are often used and they normally have a simple structure. On the other hand the library contains different components of the double clutch transmission.

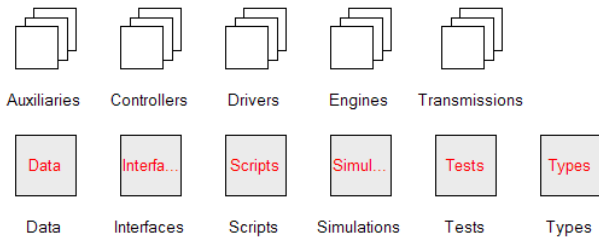


Figure 1: Library base structure

Here, it has to be differentiated between physical components of the transmission and components for control. A couple of models exist with different levels of detail or with a different implementation, e. g., gearbox with or without losses, or different controls for shifting. Furthermore, the library contains additional components for modelling, control, and simulation of the power train with a double clutch transmission. An example is the engine control in connection with the transmission control. Here, the engine torque has to be influenced by the transmission control in some parts of the shifting process for a smooth shifting.

The library is derived from the *VehicleInterfaces* library [2] and the *PowerTrain* library [3]. Consequently, the suitable interfaces and bus structures are used, so that new transmission models can be connected with the models of these libraries. Furthermore, we defined additional interfaces to get the advantage of replaceable models and blocks. Doing this, it is possible to define some basic models and structures and it is easy to build models with different levels of detail. The overall structure of the library is arranged according to the different components like controllers, engines, transmissions, auxiliaries, interfaces, etc. The library base structure is shown in Figure 1. Besides, the *VehicleInterfaces* library, the *PowerTrain* library, and the Modelica Standard Library [4] are used.

2.1 Introduction of state signals

State graphs [5] are used at important places for the control of the double clutch transmission, because the control process of the double clutch transmission depends often on more than one state. The active states of the particular state graphs have to be available through the control from different points. According to this, it is necessary to get an access to the active states. Here, we introduced so called state signals which are accessible in our case by the *TransmissionControlBus* in the *ControlBus* of the *VehicleInterfaces*. A state signal is defined by an Integer and the particu-

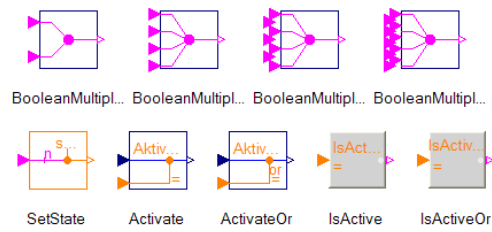


Figure 2: State signals relevant blocks

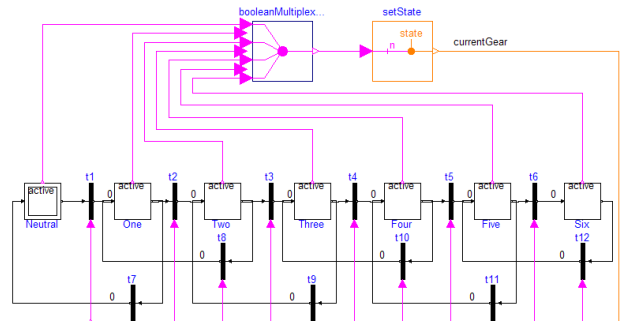


Figure 3: Generating a state signal in a state graph

lar states are represented by the present Modelica implementation of the Enumerations. Inside a state graph the particular states are combined to one state signal, which represents always the active state by the associated Integer value of the Enumeration. Additionally, we created new blocks to access these state signals so that the state signal can be compared with a desired state and a specific control can be activated.

The different blocks for using state signals are shown in Figure 2. The blocks in the first row and the block *SetState* are used to generate the desired state signal. Figure 3 shows an example for generating a state signal in a state graph. The input size of the *BooleanMultipl...* depends here on the number of states in the state graph. The block *SetState* determines the active state and assigns the accordant Enumeration value to the state signal. The code is shown in Listing 1.

There are two types of blocks to access the state signals. First, a real signal can pass or is set to zero by the blocks *Activate* and *ActivateOr*. These blocks work like a switch. Listing 2 shows the code example for the block *Activate*. Second, the blocks *IsActive* and *IsActiveOr* generate a Boolean signal according to the signal state. All four blocks are parameterised by the chosen Enumeration value. They compare the actual state signal and the Enumeration value and set the output signal accordingly. The 'or'-variants use a logical 'or' comparison to different Enumeration val-

Listing 1: Code example for SetState

```

model SetState
  extends
    Modelica.Blocks.Interfaces.IntegerSO;
  parameter Types.DCTypes.Temp
    states[:]={1,2};
  Modelica.Blocks.Interfaces.BooleanInput
    u[size(states, 1)];
  protected
    Integer sz = size(states, 1);
  algorithm
    for i in 0:sz-1 loop
      if u[sz-i] then
        y:=states[sz-i];
      end if;
    end for;
end SetState;
  
```

Listing 2: Code example for Activate

```

block Activate
  extends
    Modelica.Blocks.Interfaces.SO;
  parameter VehicleDCT.Types.DCTypes.Temp
    state(start=1);
  Modelica.Blocks.Interfaces.RealInput u;
  Modelica.Blocks.Interfaces.IntegerInput
    statesignal;
  equation
    y = if (statesignal==state)
      then u else 0;
  end Activate;
  
```

ues, respectively.

Furthermore, an advantage is the possibility to plot the state signals like other simulation variables. Thus, the change of the states of each state graph can be observed after the simulation. A possible disadvantage is the increase of variables and equations.

3 Different models of a double clutch transmission

The aim of the simulation process is to get data of shift processes of different quality. According to this, it is necessary to build models with different levels of detail. A basic model of a double clutch transmission is shown in Figure 4. It represents the base for further models by containing replaceable elements in the front and back layers using the interfaces and elements of the described library.

Figure 4 shows the two shafts with the two clutches. The first shaft contains the odd gears and the reverse gear, and the second shaft contains the even gears. The

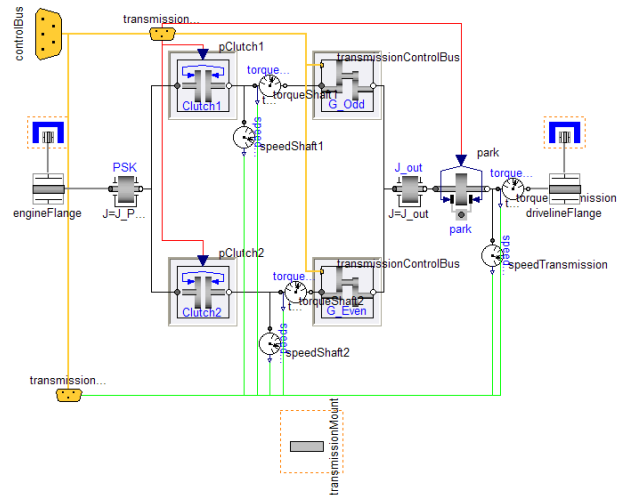


Figure 4: Basic double clutch transmission model

synchronisers of the particular gears are also modelled as clutches. Here, the transmission control has to ensure the correct gear synchronisations. Only one gear per shaft is allowed to be synchronised. The *TransmissionControlBus* contains the control signals for the two parallel clutches and for the particular synchronisers. All measured signals like revolution speeds and torques are added to the *TransmissionBus*. In a real vehicle torques are not measured, because of the difficult measurement of a dynamic torque. Normally, the torque is calculated by characteristic curves.

The replaceable elements can be modelled with different levels of detail. A clutch can be either a dry or a laminar clutch. Many components can be chosen as ideal or with losses. Furthermore, it is possible to use different numbers of spring and damper elements and different spring and damping constants.

4 Control of the transmission

The modelling of the control is just as important as the physical modelling of the double clutch transmission. The outputs of the transmission control are the control signals for the two clutches and for the synchronisers of the particular gears. Additionally, the transmission control calculates internal control signals like the actual driving state, the current gear, or the requested gear. The main focus is the control of the different shifting processes. At this point, the correct control is very important in view to smooth shiftings, because both clutches are engaged. Control errors can result in oscillations, jerks, or even in a damage of the transmission.

Figure 5 shows the structure of the transmission con-

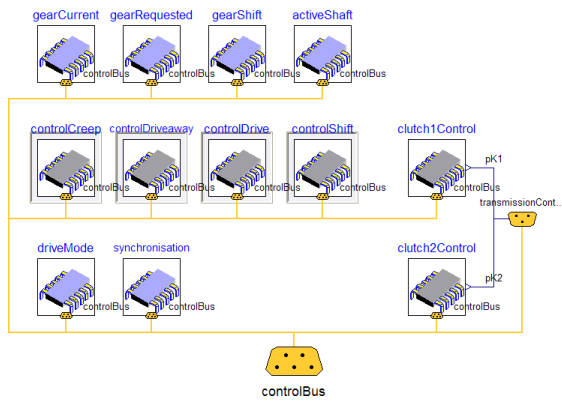


Figure 5: Transmission control model

trol. All control components extend the *Controllers* interface and use *ControlBus* sub buses of the *VehicleInterfaces*. The models *GearCurrent* and *GearRequested* determine the current and the requested gear using state graphs and assigning the states to state signals. Additionally, *DriveMode* appoints the actual driving state, e. g., creep, drive, upshift, etc., and assigns a corresponding state signal, too. The block *GearShift* identifies the type of the shifting using the drive mode and the power transmission direction of the power train. *ActiveShaft* determines the active transmission shaft and the active clutch, respectively. This information is very important in view to the control of the clutches, because the clutch control values are calculated independent of the clutch number. There is only one differentiation between the active and the inactive clutch, or during a shifting process between the engaging and the disengaging clutch. Doing this, each control value is calculated only once using an individual control block.

Currently, the clutch control distinguishes between creep, driveaway, drive, and shift. For all these drive modes exists a replaceable control. The controls calculate independent values for the active clutch. In consideration of the drive mode and the active shaft the block *ClutchXControl* assigns the control signal for the particular clutch. As shown in Figure 6, the clutch control signal is calculated using *Activate* and *ActivateOr* blocks with the state signals. Doing this, the independent control values of the different drive modes are merged to one control signal for the particular clutch. Finally, the transmission control contains the model *Synchronisation*, which controls the synchronisers of the particular gears. The control has to safeguard that only one gear per transmission shaft is synchronised.

The aim of the modelling and the simulation of a dou-

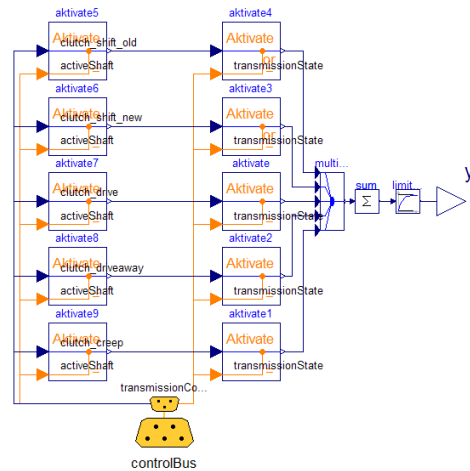


Figure 6: Clutch control model merging the particular clutch control values

ble clutch transmission are to get data of shifting processes of different quality. According to this, the focus of this work is on the control of the shifting process. The shifting process is controlled by the model *ControlShift*. There exist four different shifting types in consideration of downshift and upshift as well as pulling and pushing power of the power train. Additionally, these types are separated in two phases. The shifting types are: pull upshift, push downshift, pull downshift, and push upshift.

The structure of the model *ControlShift* is shown in Figure 7. For each shifting type and each phase exists a replaceable control using the same interface. The three output signals are *clutch_shift_old*, *clutch_shift_new*, and *engineEngagement*. The input is an activation signal in form of a Boolean signal. If the activation signal equals false, then the outputs have to be set to zero. Only one shifting process can be activated, because all blocks work in parallel. All types and phases are replaceable elements to simulate shifting processes of different quality.

The two frequently used shifting types are pull upshift and push downshift. They are similar in their control behaviour. In the first phase the torque is transmitted from the old clutch to the new clutch, so both clutches are engaged. Then in the second phase of the shifting process the engine speed is passed to the speed of the new shaft. The engine speed decreases while upshifting and increases while downshifting. Two possibilities exist for the adjustment of the engine speed. On the one hand it can be managed by an increase of the clutch capacity. Then the drive end impacts the engine torque and the engine speed decreases or increases, respectively. The clutch capacity is a function of the

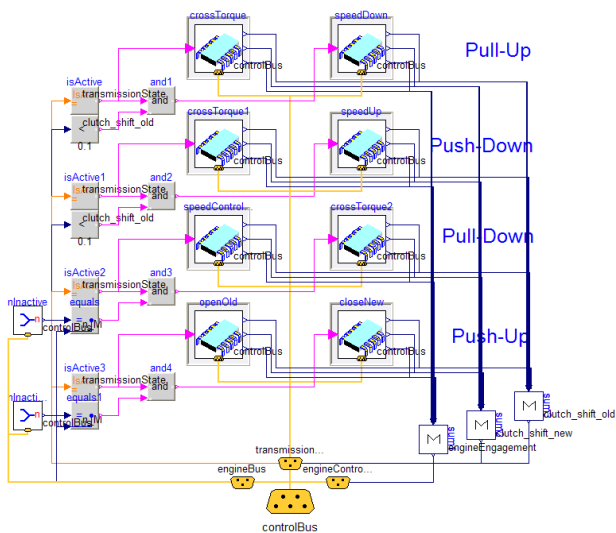


Figure 7: Shift control model with replaceable shift modes and phases

clutch engagement and describes the clutch’s maximum possible transmission torque. The real transmitted torque of the clutch is a function of the clutch capacity and the input torque. Therefore, the clutch is either in slipping or locked mode. On the other hand it is possible to do an engine engagement. This means that the engine torque increases or decreases and to that effect the engine speed will be changed. In this work we use an engine based on an engine map. According to this, an engine engagement changes the internal throttle position as input of the engine map.

The other two shifting types are pull downshift and push upshift. Here, the order of the two phases changes. At first there is the adjustment of the engine speed to the second transmission shaft speed. Finally, the torque is transmitted from the first to the second clutch. The downshift with pulling power occurs in consequence of a kick down. Then, it is possible that both clutches are engaged during the speed adjustment, because of the demand of high power transmission. An upshift with pushing power occurs normally if the vehicle rolls downhill with increasing speed and the engine speed exceeds the upshift threshold.

Now, it is possible to simulate shifting processes of different quality. A good shifting results in an uninterrupted power transmission. In this case the driver does not sense a changing of the acceleration. Then the sum of both clutch torques equals the engine torque less the dynamic torques at any time. A bad shifting in consequence of control errors can result in revolution speed droppings, break outs, or oscillations. This leads to a changing of the acceleration and will be sensed by the

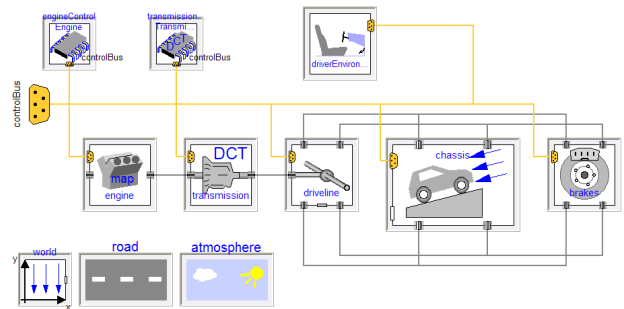


Figure 8: Vehicle model with replaceable elements

driver.

5 Simulation of shifting cycles

With the described models and controls, simulations of different shifting cycles are executed. By varying the driver models it is possible to drive miscellaneous drive and shifting cycles, e. g., in form of a table with acceleration and brake pedal positions, or following a velocity profile in form of predefined drive cycles. An example for a drive cycle is the New European Driving Cycle NEDC [6].

The vehicle simulation structure is shown in Figure 8. The components are replaceable or have internal replaceable elements. Furthermore, it is possible to use existing models, e. g., of the *PowerTrain* library. Both the redeclaration of physical components and the shift control can be changed. Thus, the simulations generate data of different quality due to the choice of the grade of the control.

6 Results and application

Based on the presented vehicle simulation structure we can now simulate drive and shifting cycles. As an example, the simulation results of a drive cycle are shown in Figure 9. The drive cycle contains the following sections: engine start, creep, driveaway, upshifts, drive, downshifts, and stoppage. The upshifts are realised with an engine engagement and the downshifts are realised with an increase of the clutch capacity for the adjustment of the engine speed.

At the top the Figure 9 shows the speeds of the engine and of the two transmission shafts. In the next part the capacity of the two clutches and the engine torque can be seen. The up- and downshifts can be easily recognised by the crossing of the clutch capacities and the change of the engine speed between the

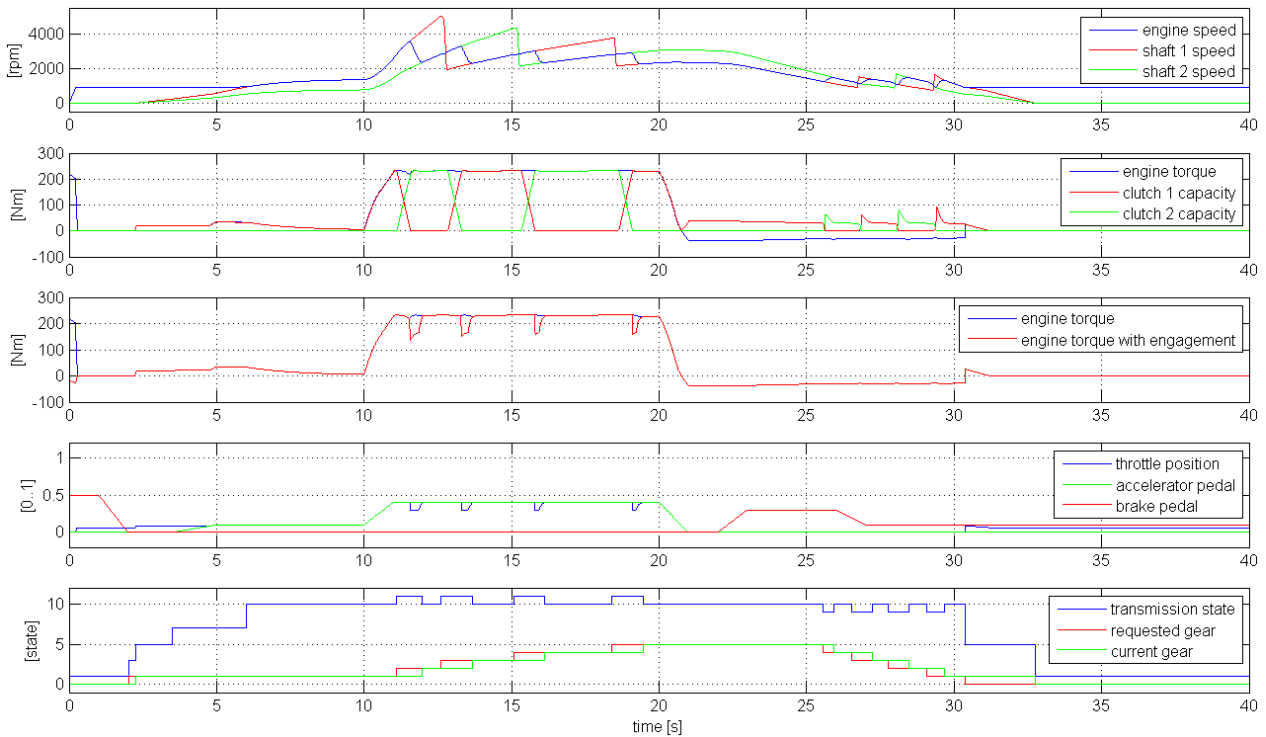


Figure 9: Simulation results of a simple drive cycle

transmission shaft speeds. During the downshifts the clutch capacity increases for the adjustment of the engine speed as described in Section 4. The third part shows the engine engagement at the upshifts. At the upshift the engine torque is decreased during the engine speed adjustment to the new transmission shaft speed. The engine control decreases the throttle position on demand of the transmission control to decrease the engine torque. The acceleration pedal, the brake pedal, and the decreased throttle position can be seen in the fourth plot of Figure 9. The throttle position increases at the start and at the end of the simulation, because of the engagement of the engine speed governor. The bottom of Figure 9 shows the transmission state according to the described sections. Furthermore, the current and the requested gear can be seen. These three signals are state signals and the Integer value represents a particular state.

In the past, we developed in cooperation with the IAV GmbH an automated measurement data analysis system [1]. The evaluation process of the software can be parameterised efficiently by XML templates. The system supports the common automotive measurement file formats and it can handle huge data traces with a sequential data processing. Moreover, there exist several intelligent signal processing modules for the data evaluation.

For complex analyses the system parametrisation is

difficult, because the measurement data is often unlabelled. The data contains measurements of several shifting processes, but there is normally no information about good or bad examples. Now, we have the possibility to simulate shifting processes of different quality. With the simulation results of this work we can parameterise the automated measurement data analysis system for the evaluation of shifting processes. The aim of the measurement data analysis is the detection and assessment of bad shifting in the vehicle development.

7 Conclusion and outlook

In this paper, we presented the modelling, control, and simulation of a double clutch transmission. The focus is the simulation of data of different quality particularly with regard to shifting processes. According to this, we developed model and control structures with replaceable elements. It is possible to use the same vehicle model for the simulation of good and bad shiftings. On the one hand the level of detail can be changed. On the other hand it is possible to redeclare elements of the transmission control to change the control behaviour.

We presented a library with basic transmission components and introduced state signals to access sys-

tem states from different points of the control process. Furthermore, we described the developed transmission control and discussed the different shifting types in detail. The vehicle model can be used to simulate a power train with a double clutch transmission. Finally, the simulation results of a simple drive cycle were shown.

With the presented models we get the background for future works. We developed a base model of a double clutch transmission and a base control structure with a working control for the four basic shifting types. At the moment there exist basic controls for each phase of the shifting. The next step will be the variation of the control to get data of different quality. Additionally, we will simulate drive cycles using double clutch transmission models of different levels of detail. With these results we will parameterise the automated measurement data analysis system for the evaluation of shifting processes of double clutch transmissions in real vehicle measurement data.

References

- [1] Henrik Isernhagen, Helmut Neemann, Steffen Kühn, and Clemens Gühmann. Intelligent signal processing in an automated measurement data analysis system. In *Proceedings of the IEEE Symposium on Computational Intelligence in Image and Signal Processing (CIISP 2007)*, pages 83 – 87, Honolulu, USA, April 2007.
- [2] Mike Dempsey, Magnus Gäfvert, Peter Harman, Christian Kral, Martin Otter, and Peter Treffinger. Coordinated automotive libraries for vehicle system modelling. In *Proceedings of the 5th International Modelica Conference*, pages 33–41, Vienna, September 2006. The Modelica Association and arsenal research.
- [3] Jakub Tobolář, Martin Otter, and Tilman Bunte. Modelling of Vehicle Powertrains with the Modelica PowerTrain Library. In *Systemanalyse in der Kfz-Antriebstechnik IV*, volume 79 of *Haus der Technik Fachbuch*, pages 204–216, Augsburg, February 2007.
- [4] Modelica. <http://www.modelica.org>.
- [5] M. Otter, K.-E. Årzén, and I. Dressler. StateGraph - A Modelica Library for Hierarchical State Machines. In *Proceedings of the 4th International Modelica Conference*, pages 569–578, Hamburg, March 2005. The Modelica Association and the Hamburg University of Technology.
- [6] Emission Test Cycles. Summary of worldwide engine and vehicle test cycles. <http://www.dieselnet.com/standards/cycles/>.

TestWeaver

A Tool for Simulation-based Test of Mechatronic Designs

Andreas Junghanns, Jakob Mauss, Mugur Tatar
 QTronic GmbH, Alt-Moabit 91d, D-10559 Berlin
 {andreas.junghanns, jakob.mauss, mugur.tatar}@qtronic.de

Abstract

The tight interaction among an ever increasing amount of software functions and hardware subsystems (mechanics, hydraulics, electronics, etc.) leads to a new kind of complexity that is difficult to manage during mechatronic design. System tests have to consider huge amounts of relevant test cases. Validation with limited resources (time and costs) is a challenge for the development teams. We present a new instrument that should help engineers in dealing with the complexity of test and validation. TestWeaver is based on a novel approach that aims at maximizing test coverage with minimal work load for the test engineer for specifying test cases. The method integrates simulation (MiL/SiL) with automatic test generation and evaluation, and has found successful applications in the automotive industry. We illustrate the approach using a 6-speed automatic transmission for passenger cars. We present also the way TestWeaver and Modelica simulators can work together.

Keywords: test automation, mechatronic systems

1 Introduction

When developing complex mechatronic systems, like a hybrid drive train or an automatic transmission for a vehicle, contributions from different engineering disciplines, design teams, departments, and organizations have to be integrated, resulting in a complex design process. Consequently, during development, design flaws and coding errors are unavoidable. For an OEM, it is then crucial that all those bugs and weak points are found and eliminated in time, i.e. before the system is produced and delivered to customers. Failing to do so may result in expensive recalls, high warranty costs, and customer dissatisfaction. OEMs have long realized this and spend up to 40% of their development budgets for test related activities. Software offers great flexibility to implement new functions, but also many hidden opportunities to introduce bugs that are hard to

discover. Moreover, the complex behaviour that results from the interaction of software and physical systems cannot be formally and completely analysed and validated. Most often, it can only be evaluated in a limited amount of points with physical or virtual experiments. The development teams are often faced with a dilemma: on the one side, the system test should cover a huge space of relevant test cases, on the other, there is only a very limited amount of available resources (time and costs) for this purpose.

This paper presents a novel test method that has the potential to dramatically increase the coverage of testing without increasing the work load for test engineers. We achieve this by generating and executing thousands of tests automatically, including an initial, automated assessment of test results. The test generation can be focused on certain state spaces using constraints and coverage goals.

This paper is structured as follows: In section 2, we take a bird's-eye view on testing mechatronic systems. In section 3, we survey the main test methods used in the automotive industry today. Section 4 presents the proposed test method. Section 5 illustrates the proposed method using a 6-speed automatic transmission for passenger cars. Section 6 discusses our approach to automated test evaluation. We conclude the paper with a summary of the benefits of our test method, and discuss its applicability to other engineering domains.

2 The challenge of testing

When testing a mechatronic system, it is usually not sufficient to test the system under laboratory conditions for a couple of idealized use cases. Instead, to increase the chance to discover all hidden bugs and design flaws, the system should be tested in as many different relevant conditions as possible. Consider as an example an assembly such as an automatic transmission used in a passenger car.

In this case, the space of working conditions extends at least along the following dimensions:

- *weather*: for example, temperatures range from - 40°C to 40°C, with significant impact on oil properties of hydraulic subsystems
- *street*: different road profiles, uphill, down hill, curves, different friction laws for road-wheel contact
- *driver*: variations of attitude and behavior of the human driver, including unforeseen (strange) ways of driving the car
- *spontaneous component faults*: during operation, components of the assembly may spontaneously fail at any time; the control software of the assembly must detect and react appropriately to these situations, in order to guarantee passenger safety and to avoid more serious damage
- *production tolerances*: mechanical, electrical and other physical properties of the involved components vary within certain ranges depending on the manufacturing process
- *aging*: parameter values drift for certain components during the life time of the assembly
- *interaction with other assemblies*: a transmission communicates with other assemblies (engine, brake system) through a network that implements distributed functions; for example, during gear shifts, the transmission might ask the engine to reduce the torque in order to protect the switching components.

These dimensions span a huge space of possible operational conditions for an assembly. The possibilities along each dimension multiply to form a huge cross-product. The ultimate goal of testing is to verify that the system performs adequately at every single point of that space. It would be great to have techniques to mathematically prove certain properties of the system (such as the absence of unwanted behavior), which would enable a test engineer to cover infinitely many cases within a single work step. However, such proof techniques (e.g. model checking, cf. [1]) are by far too limited to deal with the complexity of the system level test considered here.

In practice, the goal of covering the entire state space is approximated by considering a finite number of test cases of that space.

3 A critical view on some test methods in use

Testing at different functional integration levels (e.g. component, module, system, vehicle) and in different setups (e.g. MiL, SiL, HiL, physical prototypes) is nowadays an important, integral part of the development process. The earlier problems are discovered and eliminated, the better. Very often, however:

- (a) relevant tests can only be formulated, or have to be repeated, at higher levels of functional integration (e.g. at system level) - consider, for instance, the system reaction in case of component faults
- (b) system-level tests are only performed in a HiL or physical prototype setup¹.

Let us briefly review some of the limitations of the HiL- / physical prototype based testing:

- *time, costs, safety*: physical prototypes and HiL setups are quite expensive and busy resources; testing takes place late in the development cycle; not too many tests can be conducted; the reaction to certain component faults cannot be tested with physical prototypes due to safety hazards
- *lack of agility*: it usually takes a long time between the change of a software function and the test of its effects
- *limited precision or visibility*: due to real-time requirements the physical system models used in HiL setups are often extremely simplified and therefore extremely imprecise; debugging and inspection of hidden system properties is difficult if not impossible for these setups.

Note, the above limitations are not present in MiL / SiL setups. While the importance of the HiL tests and of the tests based on physical prototypes should not be underestimated, our argument here is that they must be complemented by a more significant role of MiL and SiL tests at system level. See also [4], [5].

Irrespective of the setup used, the main limitation of common system-level test practices is the limited test coverage that can be achieved with reasonable effort. For example, test automation in a MiL / HiL setup is typically based on hand-coded test scripts for stimu-

¹ There are several reasons why this is very often the case. An important one stems from the complexity of the mechatronic development processes: several disciplines, several teams, several tools, several suppliers – together with lacking standards and practices for exchanging and integrating executable functional models.

lating the partially simulated assembly with a sequence of test inputs, including code for validating the measured response. Coding and debugging such test scripts is a labor intensive task. Given typical time frames and man power available for testing, only few (say a few dozen) cases from the huge space of possible use cases can be effectively addressed by such a script-based approach. For testing using a test rig or by driving a car on the road, this figure is even worse. For example, it is practically impossible to systematically explore the assembly's response in the case of single component faults in a setup that involves dozens of physical (not simulated) components.

When testing for the presence (or absence) of a certain system property, script-based tests verify such a condition only during a few, specifically designed scenarios and not throughout all tests.

In practice this means that many scenarios are never explored during system test and that for those scenarios explored, usually only a few of the relevant system properties are tested. Consequently, bugs and design flaws may survive all tests. These are risks the method presented here can help to reduce, adding additional robustness to the design process.

4 Exploring system behavior with TestWeaver

TestWeaver is a tool supporting the systematic test of complex systems in an autonomous, exploratory manner. Although the method could, in principle, be applied to HiL setups as well, it is primarily geared towards supporting the MiL and SiL setups. The overall design objectives of TestWeaver were to:

- (a) dramatically increase the test coverage, with respect to system behavior, while
- (b) keeping the workload for the test engineer low.

To achieve this, we wanted to remove the necessity of exclusively relying on hand-coded test scripts, since we had identified script production as the main hindrance on the way towards broad test coverage.

4.1 The “chess” principle

The key idea was: Testing a system under test (SUT) is like playing chess against the SUT and trying to drive it into a state where it violates its specification. If the tester has found a sequence of moves that drives the SUT in such an unwanted state, he has

won a game, and the sequence of moves represents a failed test.

There are more analogies: To decide for a next best move, chess computers just explore recursively all legal moves possible in the current state and test whether these lead to a goal state. This search process generates a huge tree of alternative (branching) games. In TestWeaver, the automated search for bugs and design flaws is organized quite similarly. Our method assumes that the SUT is available as an executable simulation (MiL) or as a co-simulation of several modules (SiL). As usually done, the SUT is augmented with a few components that communicate with the test driver. These communication components, called *instruments*, implicitly carry the “rules of the game” that TestWeaver is “playing” with the instrumented SUT. Namely, they carry information about: the control actions that are legal in a certain situation, the interesting qualitative states reached by the SUT, and, eventually, the violation of certain system requirements. Each instrument specifies a (relevant) dimension of the SUT state space. The value domain along each dimension has to be split into a finite set of partitions. Each SUT, or SUT-module, has to be configured individually by placing and parameterizing the instruments inside the SUT. The “game” is played in this multi-dimensional partitioned system space.



Figure 1: The chess principle

4.2 Instruments

An instrument is basically a small piece of code added to the un-instrumented version of the SUT using the native language of the executable, e.g. Modica, Matlab/Simulink, or C. The instruments communicate with TestWeaver during test execution, which enables TestWeaver to drive the test, to keep track of reached states, and to decide during test execution whether an undesired state (failure) has been reached. TestWeaver supports basically two kinds of instruments, *action choosers* and *state reporters*, that can come in several flavors:

1. state reporter: this instrument monitors a discrete or a continuous variable (e.g. a double) of the SUT, and maps its value onto a small set of partitions or discrete values (e.g. low, medium, high). During test, this instrument reports each partition change of the monitored variable to TestWeaver. This is used by TestWeaver to keep track of reached states and to maximize the coverage of the partitioned / discrete state space.

2. alarm reporter: this is actually a state change reporter. In addition the partitions are associated with severity levels, such as nominal, warning, alarm, and error. The reachability of a “bad” state corresponds to a failure of the test currently executed. Note: these failure conditions are verified throughout all the tests run by TestWeaver.

3. action chooser: this instrument is associated with an input variable of the SUT. In an automotive application, an input variable may represent the acceleration pedal or the brake pedal of a car. Depending on the details of the instrumentation, this instrument asks TestWeaver either periodically or when a trigger condition becomes true to choose a discrete input value for its input variable from the partitioned value domain of the variable.

4. fault chooser: this is a special case of action chooser. The value domain is partitioned into nominal and fault partitions and can be used to represent alternative fault modes of a component of the SUT. For example, a shift valve model may have behavior modes such as: ok, stuckClosed, and stuckOpen. Instruments like these are used by TestWeaver to inject (activate) a component fault occurring spontaneously during test execution.

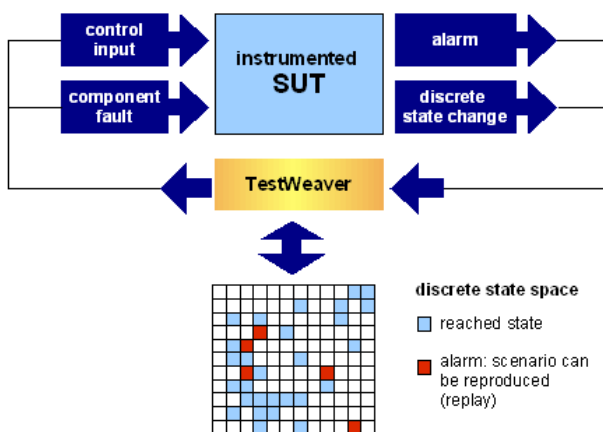


Figure 2: Instruments connect SUT to TestWeaver

Engineers like to work with their favorite modeling environment. Therefore we have implemented versions of the above instruments for alternative modeling environments, including Matlab/Simulink, Mod- elica, and C. The idea is to allow the test engineers to

instrument a SUT in their favorite modeling lan- guage, i.e. using the native implementation language of the SUT, or of the SUT-module that they are working on.

In addition to the explicit instruments, TestWeaver monitors the process of executing the SUT and records problems, such as divisions by zero, memory access violations, or timeouts in the communication.

4.3 Experiments, scenarios and reports

In TestWeaver, an *experiment* is the process of exploring and documenting the states reached by the SUT during a certain period of time, possibly taking into consideration additional search constraints and coverage goals. An experiment usually runs completely autonomously for a long time, typically several hours, and without requiring any user interaction.

When running an experiment, TestWeaver generates many differing scenarios, by generating differing sequences of answers for the action choosers. A *scenario* is the trace (or protocol) of a simulation run of the given SUT in the partitioned state space. TestWeaver combines several strategies in order to maximize the coverage of the reached system states and to increase the probability of finding failures. The results are stored in a *scenario data base* of the experiment, i.e. a tree of scenarios (actually a directed graph), as shown in Fig. 3.

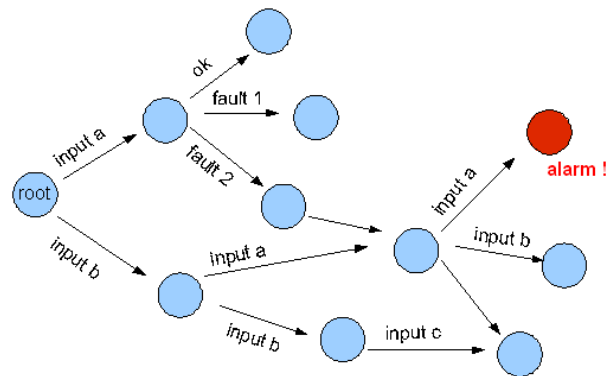


Figure 3: Scenarios generated by an experiment

The user can investigate the states reached in an experiment using a high level query language similar to SQL. Results are displayed in reports. A *report* is basically a table that displays selected properties of the scenarios stored in the scenario data base. The user specifies the structure and layout of a table by templates, while the content of a table depends on the content of the scenario data base – see Fig. 9. There are two kinds of reports: *overview reports*, that

document state reachability, and *scenario reports*, that document details of individual scenarios.

A user may specify, start, and stop an experiment, reset the experiment's data base and investigate the reports generated by the experiment, the last even while the experiment is running. Individual scenarios can be *replayed*: i.e. the SUT is restarted and is fed with the same sequence of inputs as the one recorded in order to allow detailed debugging of a problem, e.g. by plotting signals and other means.

4.4 The experiment focus

The dimensions and the partitions of the state space are configured by the instruments of the SUT. Apart of these there are also other means that can constrain the exploration, either as part of the instrumented SUT, or as explicitly defined in the specification of the *experiment focus* in TestWeaver. The focus of an experiment specifies which region of the state space should be investigated when running the experiment. During an experiment, TestWeaver tries to drive the SUT into those states that are in the experiment's focus. The experiment focus is currently specified using two means:

- *constraints*: the constraints limit the size of the considered state space of an experiment. They can limit, for instance, the duration of a scenario, or the allowed combinations of inputs and states. A high level constraint language is provided for this purpose. In an automotive application, a user could, for example, exclude all scenarios where brake pedal and acceleration pedal are engaged simultaneously. For a fault analysis, a constraint could be used to exclude certain fault modes from investigation, or to limit the number of faults inserted in a scenario: typical values are 0, 1 and 2. Higher numbers are reasonable when investigating fault-tolerant systems, e.g. systems with complex fault detection and reconfiguration mechanisms
- *coverage*: the user can tell TestWeaver to use some of the reports of the experiment as defining the coverage goals of the experiment. A report used in this way is called coverage report.

Experiments with different SUT versions and with different focus specifications can be created, run and compared with each other.

4.5 Analyzing and debugging problems

The alarm and error states of the SUT are reported in the overview reports. For each problem one or more scenarios that reach that state can be recalled from the scenario database. The scenarios can be once again *replayed* and additional investigation means can be connected. Depending on the SUT simulation environment these can be, for instance: plotting additional signals, connecting additional visualization means such as animation, setting breakpoints, and even connecting to step-by-step evaluation with source code debuggers – for instance for SUT modules developed in C.

4.6 The Modelica instrumentation library

In this section we briefly present the Modelica instrumentation library of TestWeaver, cf. Fig. 4.

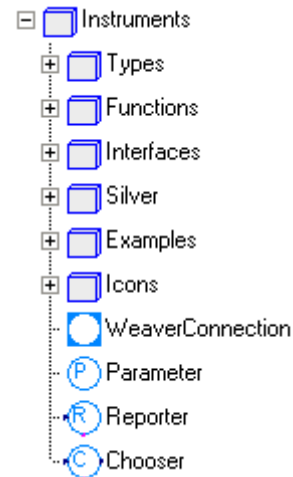


Figure 4: The Modelica Instruments library

The package *Instruments* contains ready to use components for instrumenting Modelica models. Each instrument is a Modelica component that communicates with TestWeaver through an outer *WeaverConnection*. Instruments extend the partial base class *Instruments.Interfaces.Instrument*, which is a model with the following parameters:

- *variable*: the name of the SUT variable defined by the instrument
- *comment*: a description of the instrument
- *unit*: of the classified real value, e.g. "km/h", "kg", or "%"
- *intervals*: an array of number pairs defining the numeric partitions of the real value, e.g. [0,100; 100,150; 150,1e10]
- *labels*: an array with the partition names, e.g. {"ok", "hot", "damaged"}
- *weaverConnection*: outer reference.

Furthermore, the *Instruments* package provides the following instruments as Modelica components:

- *Reporter*: an instrument that adds a severity rating to each interval and reports the value of a SUT variable when triggered; whether the reported value represents an alarm or a nominal state depends on the associated severity
- *Chooser*: an instrument that adds an occurrence rating to each interval and allows TestWeaver to control a variable of the SUT, when triggered; whether the control value represents a fault to be injected or a nominal value depends on the corresponding occurrence rating
- *Parameter*: a *Chooser* whose control variable is a model parameter to be initialized by TestWeaver at simulation start; this is used to model parameter deviations, for instance, due to aging, or due to production tolerances.

Fig. 5 shows how to use a *Reporter* in a model.

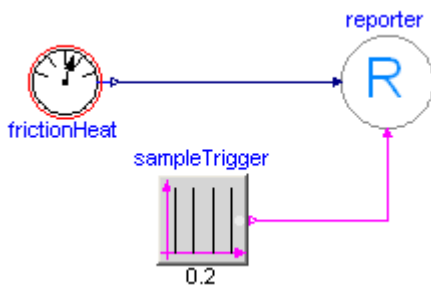


Figure 5: Reporting a temperature

The reporter is connected to the output of a heat model, and to a boolean trigger signal. Whenever the trigger signal becomes true, the reporter classifies the temperature w.r.t the partition margins and reports the result through the *WeaverConnection* to TestWeaver.

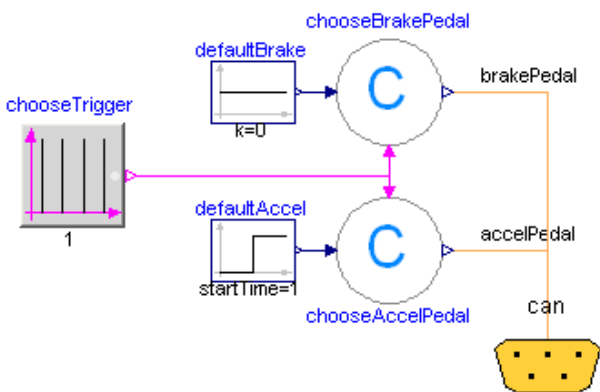


Figure 6: Controlling two pedals in a car

Likewise, Fig. 6 shows two *Choosers* called *chooseBrakePedal*, *chooseAccelPedal*. Both accept a default value at the left side, output the signal chosen by TestWeaver and accept a boolean trigger signal as input, which defines the time points when new values can be changed by TestWeaver.

5 Example: automatic transmission

As an application example for TestWeaver, consider the development of the control software for an automatic transmission. An instrumented Modelica model of an entire car, including the transmission and a *WeaverConnection* is shown in Fig. 7.

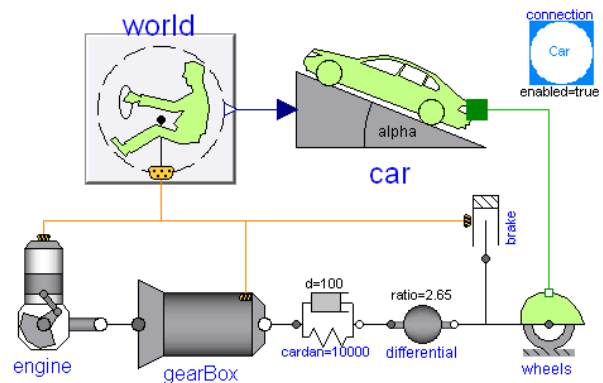


Figure 7: An instrumented car model

The control software is developed using a SiL development environment. The executable SUT is hence co-simulating two modules: the compiled control software, and the compiled Modelica model of the car. Since the Modelica model has been instrumented, the SUT also contains functions to communicate with TestWeaver. When TestWeaver starts the SUT to perform a system test, all contained instruments register themselves at TestWeaver with all their declared static properties (intervals, labels, severities etc.). TestWeaver then displays a list of these instruments, see tree in Fig. 9. Selecting an instrument in the tree displays all its properties. Fig. 8. shows how TestWeaver displays the heat reporter of Fig. 5.

Instrument heatA

Instrument role: ALARM
 Instrument type: REAL
 Instrument unit: J
 Declared by: Car

partitions	occurence	severity	color	values
damaged	10	10	#F03939	80000.0 : 1.0E10
ok	10	0	#AFDFDF	-0.1 : 40000.0
hot	10	0	#7FBFBF	40000.0 : 80000.0

Figure 8: Reporter (Fig. 5) displayed by TestWeaver

The tree shown in Fig. 9 also contains an item for each report of the experiment. Selecting such a report displays the report as table. Fig. 9. shows a report that shows which gear shifts have already been reached during the experiment, and whether critical temperatures at the clutches A and B have been reported. For each state, up to two scenarios are referenced in the right most column. Clicking on such a reference displays that scenario as a sequence of discrete states. It is possible to reproduce the entire scenario with an identical simulation such that the test engineer can access all its details. For example, run-time exceptions of the control software (such as division by zero) can be reproduced this way and inspected using the usual software debugging tools.

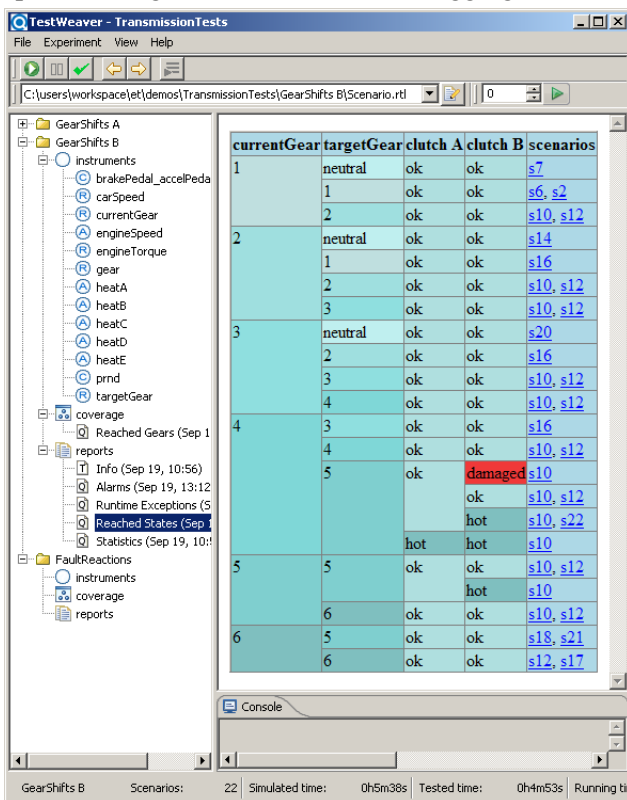


Figure 9: TestWeaver displaying a coverage report

6 The instrumentation process and quality watchers

If an executable SUT exists it is relatively easy to add the instrumentation for the action choosers and the pure state reporters – those reporters that do not attempt to classify the states as “good” or “bad”. A good understanding of the system functionality is required, in order to select the relevant system features and the relevant qualitative partitions. This activity is not completely new for the test engineers: the commonly used “Classification Tree Method” for

specifying tests relies exactly on this kind of system analysis [2].

For the automatic test evaluation TestWeaver uses alarm reporters. The alarm reporters monitor correctness or quality conditions of the SUT. Some of these conditions are easy to define, others might require quite sophisticated watchers. Let us begin with the easier cases:

- often physical components have well known functional or safety ranges of operation that should not be exceeded; examples: maximal power dissipated by a component, maximal temperature, maximal pressure in a container, maximal allowed rotational speed of an engine or gear, etc.
- similar “local” correctness conditions exist for software functions; in the software industry such *assertions* are widely used today for detecting, diagnosing and classifying programming errors during test; examples here: maximum number of allowed objects in a buffer, access indices within array bounds, assumed domains for input parameters, assertions about function pre-conditions, post-conditions and other invariants that can be easily declared and monitored – see also [3].

The above quality conditions can be locally implemented, e.g. in a component type library. In such a case, each component instantiation will also instantiate the check of the quality condition and the instrumentation required.

In addition, the run time exceptions of the SUT process (e.g. divisions by zero, memory access violations, etc.) and the communication timeouts (possibly produced by infinite loops, non-converging numerical solvers, etc.) are anyway monitored and reported by TestWeaver.

An important class of quality conditions can be relatively easily defined when the SUT includes a controller and a model of the controlled system. Often controllers have in some form (at least a simplified) inverse model of the controlled system. This makes it easier to formulate system invariants or quality conditions. For instance: when not shifting, the controller gear should match the gear from the transmission model; if no fault is set in the hardware model, the on-board diagnosis should not detect any fault; if a fault code is generated, then the fault should coincide with a fault set in the hardware model. In general, the assumptions made by the control system about the state of the controlled system should match (within certain acceptable delays and tolerances) the state of the model.

The more we migrate from checking correctness to checking quality, the more complex and subtle the watchers can become. For TestWeaver arbitrarily complex quality watchers can be implemented with Modelica, Matlab/Simulink or C. In principle, any conventional test case can be turned into a quality watcher, although it might be sometimes difficult to generalize the specific conditions checked in a test case. The effort will be rewarded because:

- (a) the quality condition will be checked not only for one input sequence, but for many differing scenarios, and
- (b) a more general formulated condition is likely to survive unchanged, or with only small changes, when new SUT versions are produced later on, during development.

The tuning of some complex quality watchers can be quite laborious. In practice, there will always be cases when false alarms are generated. Therefore, after each experiment, also a detailed manual analysis and diagnosis of the problems found is prescribed by our test method.

7 Summary and conclusions

The increasing pressure to shorten and cheapen development for more and more complex products requires new test strategies. Today we see early module tests and late system-level tests, like HiL and test-rigs, as state of the art. The importance of early system-level testing increases with the increasing complexity of module interaction because bugs on system-level are more likely, more costly to fix and harder to find. Testing before physical prototypes exist, for both controllers and hardware, is one necessary step towards early system-level testing.

As long as the behavior of a system can be described easily using stimuli-response sets, script-based testing is a feasible strategy. With increasing system complexity, this method fails to provide the necessary coverage at reasonable cost. On the other side, our test method allows to:

- (a) systematically investigate large state spaces with low specification costs: only the rules of the “game” have to be specified, not the individual scenarios
- (b) discover new problems that do not show up when using only the predefined test scenarios prescribed by traditional test methods; TestWeaver can generate thousands of new, qualitatively differing tests, depending on the time allocated to an experiment

- (c) increase the confidence that no hidden design flaws exist.

In chapter 5, we have sketched the application of TestWeaver to a SiL-based system test of an automatic transmission. We have several years of experience with this kind of applications. However, the application of TestWeaver to other domains seems promising as well, especially for those cases where a complex interaction between the software and the physical world exists. For instance:

- *driver assistance systems*: in car systems such as ABS, ESP, etc. we meet a complex interaction among the control software, the vehicle dynamics and the human driver; this leads to myriads of relevant scenarios that should be investigated during design
- *plant control systems*: in plants for chemical processes, power plants etc. we meet the interaction of the control software, plant physics and the actions of the operators; again, the same kind of complexity that calls for a systematic investigation during design.

TestWeaver runs on Windows platforms. It is a powerful, yet easy to use tool: users can use their native specification or modeling environment and don't have to learn yet another test-specification language.

Acknowledgments

Special thanks to Volker May who initiated the research that finally led to TestWeaver.

References

- [1] Berard et. al.: Systems and Software Verification: Model-Checking Techniques and Tools, Springer Verlag, 2001.
- [2] Grochtmann, M., Grimm, K.: Classification Trees for Partition Testing. Software Testing, Verification & Reliability, Volume 3, No 2, pp. 63-82, 1993.
- [3] Meyer, Bertrand: Applying "Design by Contract", in Computer (IEEE), 25, 10, October, pages 40-51, 1992.
- [4] Rebeschief, S., Liebezeit, Th., Bazarsuren, U., Gühmann, C.: Automatisierter Closed-Loop-Testprozess für Steuergerätefunktionen. ATZ elektronik, 1/2007 (in German).
- [5] Thomke, Stefan: Experimentation Matters: Unlocking the Potential of New Technologies, Harvard Business School Press, 2003.

Session 3d

Electric Systems & Applications

Simulation of Electrical Rotor Asymmetries in Squirrel Cage Induction Machines with the ExtendedMachines Library

Christian Kral Anton Haumer
Arsenal Research
Giefinggasse 2, 1210 Vienna, Austria
christian.kral@arsenal.ac.at

Abstract

In this paper a physical model of a squirrel cage induction machine with rotor topology model is presented. The parameters of the induction machine are discussed and the issue of rotor fault detection is addressed. For a machine with one broken rotor bar a Modelica simulation model is compared with measurement results. *Keywords: Electric machines, squirrel cage, rotor asymmetries*

1 Introduction

The squirrel cage of an induction machine consists of N_r bars and two end rings, connecting the bars on both ends, as depicted in Fig. 1. The fins located on the end rings are required to force the circulation of the air in the inner region of the machine.

In the manufacturing process it is intended to fabricate a fully symmetrical squirrel cage. Due to manufacturing problems or certain operating conditions, electrical rotor asymmetries can occur. The causes for such rotor electrical asymmetries are:

- shrink holes and voids in the aluminum of the bars or end rings,
- improper junctions of the bars and end rings,
- heavy duty start-ups that the machine is not designed for,
- thermal overloading of the machine,
- high temperature gradients, causing cracks.

Electric rotor asymmetry can be classified as rotor bar or a rotor end ring segment fault. These cases can be modeled by a ohmic resistance increase of either a rotor bar or an end ring segment.



Figure 1: Rotor cage of an induction machine

In the Modelica Standard Library (MSL) the induction machines are based on the assumption, that the number of phases is limited to three and that stator and rotor windings are fully symmetrical. Electrical rotor asymmetries can therefore not be modeled using the MSL. For modeling electrical rotor asymmetries of the squirrel cage induction machines the full topology of the rotor cage has to be taken into account. Appropriate models are provided by the ExtendedMachines Library [1].

2 Stator Winding Model

For the investigated fault cases it can be assumed that the stator winding is fully symmetrical. Additionally, it will be assumed, that number of stator phases is limited to three. In this case the stator voltage equation

can be written as

$$V_{s[i]} = R_s I_{s[i]} + L_{s\sigma} \frac{dI_{s[i]}}{dt} + \sum_{j=1}^3 L_{sm[i,j]} \frac{dI_{s[j]}}{dt} + \sum_{j=1}^{N_r} \frac{dL_{sr[i,j]} I_{r[j]}}{dt}. \quad (1)$$

In this equation $V_{s[i]}$ and $I_{s[i]}$ and $I_{r[i]}$ are the stator voltage and current and the rotor current, respectively. Due to the symmetry of the stator winding the stator resistance R_s and the stator stray inductance $L_{s\sigma}$ are symmetrical, too. The matrix of the main field inductances of the stator winding,

$$L_{sm[i,j]} = L_0 w_s^2 \xi_s^2 \cos \left[\frac{(i-j)2\pi}{3} \right], \quad (2)$$

and the matrix of the mutual coupling inductances between the stator and rotor,

$$L_{sr[i,j]} = L_0 w_s \xi_s \xi_r \cos \left[\frac{(i-1)2\pi}{3} - \frac{(j-1)2\pi}{N_r} - \gamma_m \right], \quad (3)$$

are fully symmetrical, since it is assumed that the coupling over the magnetic main field is not influenced by the rotor asymmetries. In this equation, L_0 indicates the base inductance of a coil without chording, i.e., the coil width is equal to the pole pitch. The parameters w_s and ξ_s are the number of series connected turns and the winding factor of the stator winding. The product $w_s \xi_s$ is the *effective number of turns*. The winding factor of the rotor winding

$$\xi_r = \sin \left(\frac{p\pi}{N_r} \right) \quad (4)$$

is a pure geometric factor, which is derived in [2]. In this equation, however, it is assumed that no skewing occurs [3]. The rotor angle γ_m represents the relative movement of the rotor with respect to the stator.

The effective number of turns, $w_s \xi_s$, may be determined from a winding topology, which is indicated by the begin and end location and the number of turns of the stator winding coils – as depicted in Fig. 2. Alternatively, a symmetric stator winding can be parametrized by entering the effective number of turns.

3 Rotor Winding Model

The squirrel cage rotor with N_r rotor bars can be seen as a winding topology with an effective number of

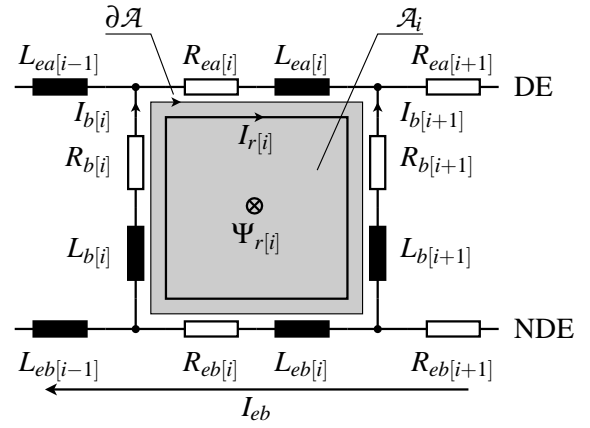


Figure 3: Topology of the rotor cage (DE = drive end, NDE = non drive end)

turns equal to one. Using the winding factor of a rotor mesh (4), the matrix of the main rotor field can be expressed as

$$L_{rr[i,j]} = L_0 \xi_r^2 \cos \left[\frac{(i-j)2\pi}{N_r} \right]. \quad (5)$$

The rotor voltage equation can be derived from the topology of the squirrel which is depicted in Fig. 3. Considering constant leakage inductances $L_{b[i]}$ and $L_{ea[i]}$ and $L_{eb[i]}$ of the bars and the end rings on both sides (index a = drive end side, DE; index b = non drive end, NDE), the rotor voltage equations yields:

$$0 = (R_{ea[i]} + R_{eb[i]} + R_{b[i]} + R_{b[i+1]}) I_{r[i]} - R_{b[i]} I_{r[i-1]} - R_{b[i+1]} I_{r[i+1]} + R_{eb[i]} I_{eb} + (L_{ea[i]} + L_{eb[i]} + L_{b[i]} + L_{b[i+1]}) \frac{dI_{r[i]}}{dt} - \frac{d}{dt} (L_{b[i]} I_{r[i-1]} + L_{b[i+1]} I_{r[i+1]} - L_{eb[i]} I_{eb}) + \sum_{j=1}^3 \frac{dL_{sr[j,i]} I_{s[j]}}{dt} + \sum L_{rr[i,j]} \frac{dI_{r[j]}}{dt} \quad (6)$$

Additional parameters of this equation are the bar resistances $R_{b[i]}$ and the resistances of the end ring segments $R_{ea[i]}$ and $R_{eb[i]}$. The topology of the rotor cage (Fig. 4) leads to $N_r + 1$ linearly independent meshes. Therefore, the mesh current I_{eb} is introduced and the additional voltage equation

$$0 = \sum_{i=1}^{N_r} R_{eb[i]} (I_{r[i]} + I_{eb}) + \frac{d}{dt} \sum_{i=1}^{N_r} L_{eb[i]} (I_{r[i]} + I_{eb}) \quad (7)$$

has to be taken into account.

It should be noted that the main field inductances $L_{ss[i,j]}$ and $L_{rr[i,j]}$ of a squirrel cage induction machine

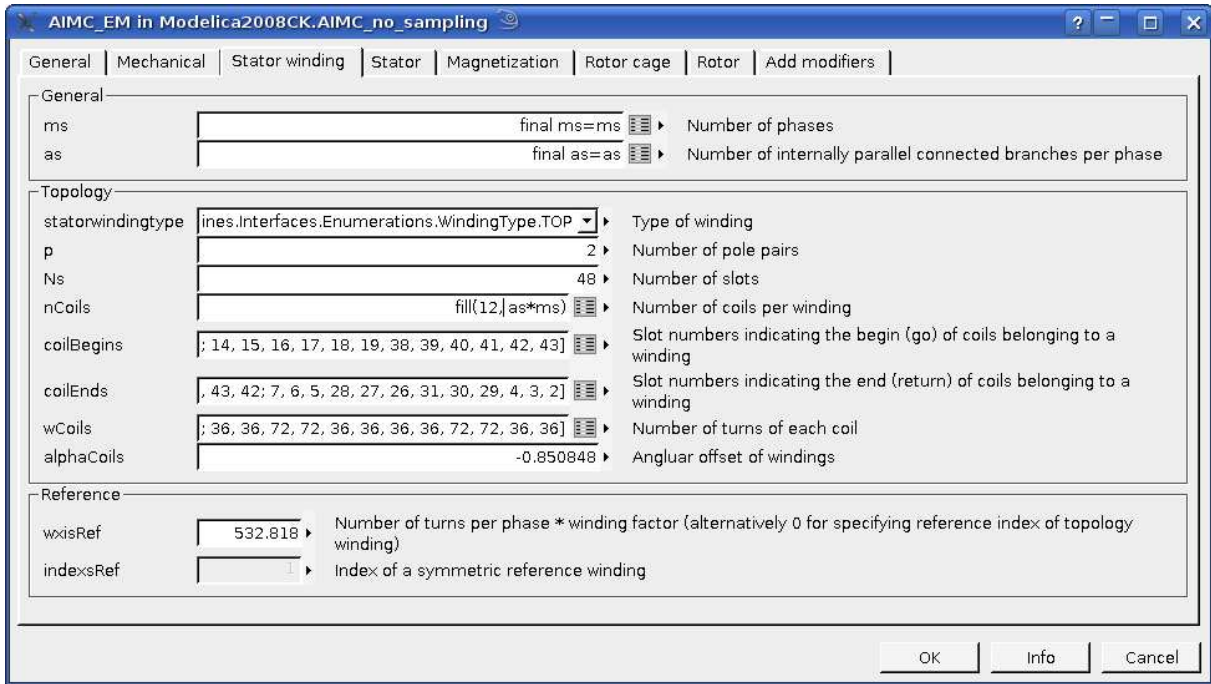


Figure 2: Stator winding parameters of a squirrel cage induction machine

are constant and the mutual inductances (3) are dependent on the rotor angle γ_m .

In the ExtendedMachines library, the rotor cage can be parametrized in two different ways. First, the resistances and leakage inductances of the rotor bars and the end ring segments of both sides can be parametrized (Fig. 4). This is how a squirrel cage is internally modeled. Second, a symmetric rotor cage can be indicated by the rotor resistance R'_r and the rotor leakage inductance $L'_{r\sigma}$, equivalently transformed to the stator side. The same parameters are used for the Machines package of the MSL. The relationship between the symmetric rotor bar and end ring resistance and the rotor resistance with respect to the stator side is determined by

$$R'_r = 2 \frac{3w_s^2 \xi_s^2}{N_r \xi_r^2} \{R_{e,\text{sym}} + R_{b,\text{sym}} [1 - \cos(\frac{2\pi p}{N_r})]\}. \quad (8)$$

A similar equation can be obtained for the rotor leakage inductance with respect to the stator side,

$$L'_{r\sigma} = 2 \frac{3w_s^2 \xi_s^2}{N_r \xi_r^2} \{L_{e\sigma,\text{sym}} + L_{b\sigma,\text{sym}} [1 - \cos(\frac{2\pi p}{N_r})]\}. \quad (9)$$

Additionally, the ratios of the resistances (ratioCageR) and leakage inductances (ratioCageL), each with respect to the rotor bars over the end ring segments, can be specified

$$\text{ratioCageR} = \frac{R_{b,\text{sym}}}{R_{e,\text{sym}}}, \quad (10)$$

$$\text{ratioCageL} = \frac{L_{b\sigma,\text{sym}}}{L_{e\sigma,\text{sym}}}. \quad (11)$$

This way, the symmetric cage resistance and leakage inductance parameters can be determined from R'_r , $L'_{r\sigma}$, ratioCageR and ratioCageL.

4 Torque

The electromagnetic (inner) torque of the machine is computed by

$$T_{\text{el}} = \sum_{i=1}^3 \sum_{j=1}^{N_r} \frac{dL_{sr[i,j]}}{d\gamma_m} I_{s[i]} I_{r[j]}. \quad (12)$$

In the presented investigation neither friction nor ventilation losses nor stray load losses are taken into account.

5 Theoretical Background of Rotor Faults

The distorted rotor bar currents have an impact on the fundamental wave of the rotor magneto motive force

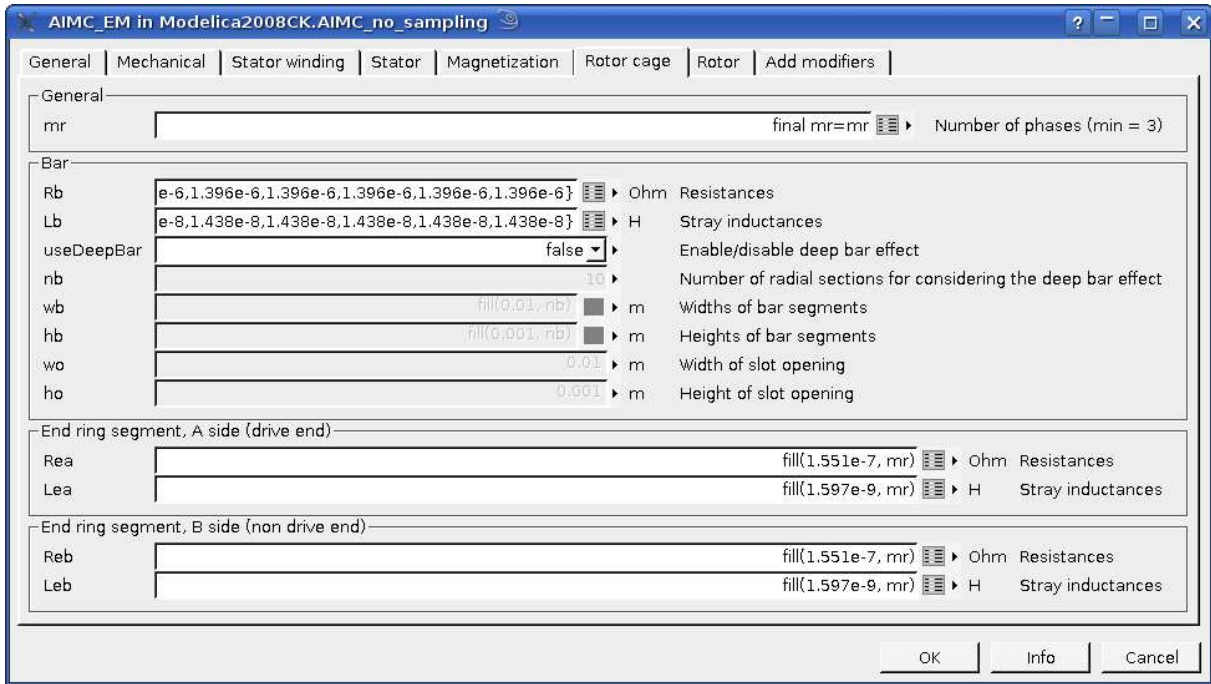


Figure 4: Parameters of the resistances and the leakage inductances of the rotor bars and end ring segments

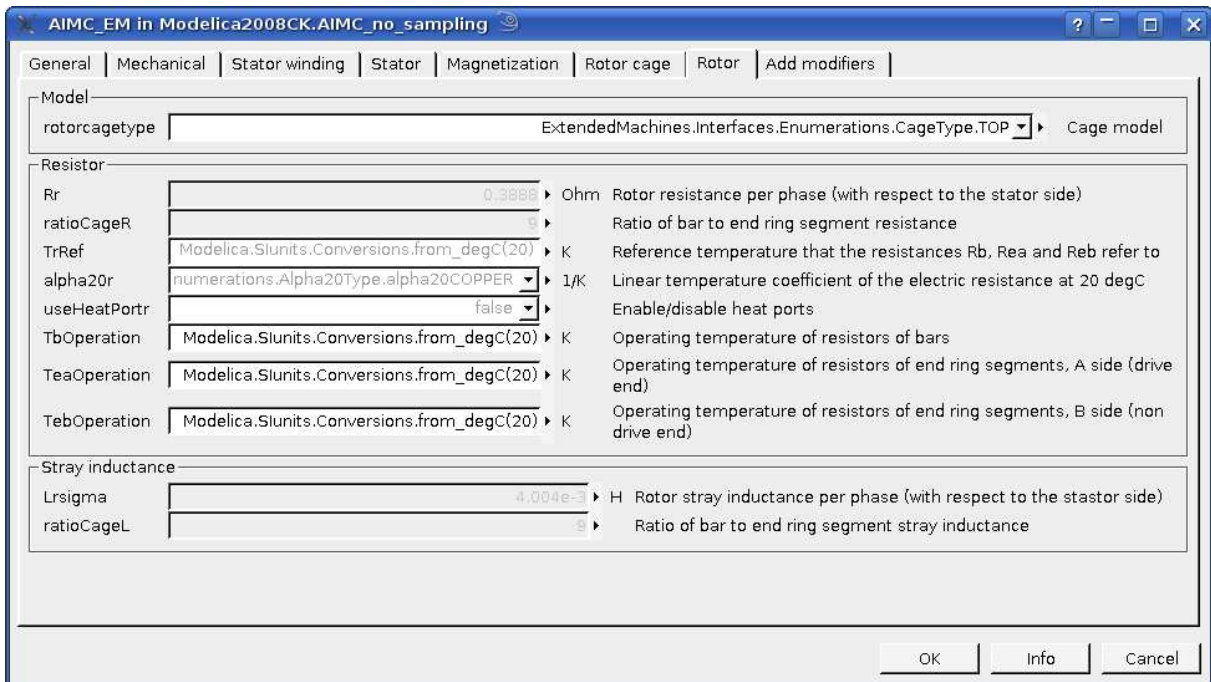


Figure 5: Parameters of the squirrel cage



Figure 6: 18.5 kW four pole induction machine

(MMF). The fundamental rotor MMF can be considered to be composed of a forward and backward traveling wave. The backward traveling wave is caused by the electrical rotor asymmetry and induces a stator voltage harmonic component at the frequency

$$f_t = (1 - 2s)f_s \quad (13)$$

for infinite inertia drives. In this equation f_s is the stator supply frequency and

$$s = \frac{f_s - pn}{f_s} \quad (14)$$

is slip, expressed in terms of rotor speed n and the number of pole pairs p . Due to the impedance of the machine (and the supply) the stator voltage harmonic gives rise to a stator current harmonic with the same frequency. To this stator current harmonic component literature refers as *lower side band* harmonic. A finite inertia of the drives gives rise to additional upper side band harmonics at the frequency

$$f_u = (1 + 2s)f_s, \quad (15)$$

the *upper side band harmonic* [4]. Between no load and rated operating conditions slip varies between zero and some per cent.

Electrical rotor asymmetries give also rise to a distortion of the magnetic field of the air gap [5] and the stray flux [6]. Additional effects are caused by the interaction of the current side band harmonics with the fundamental wave of the voltage, which gives rise to double slip frequency oscillations

$$f_t = 2sf_s \quad (16)$$

of the electrical power and torque [7], [8]. The magnitudes of these fault specific oscillations are much smaller the average values of the electrical power and torque, respectively.



Figure 7: For broken one rotor bar a hole is drilled into the aluminum part of the squirrel cage rotor

6 Investigated Machine

Simulation and measurement results refer to a 18.5 kW, four pole induction machine with 40 rotor bars (Fig. 6). In this paper simulation and measurement results are obtained for nominal load torque, nominal line-to-line voltage (400 V) and nominal frequency (50 Hz). The investigations refer to a squirrel cage with one fully broken rotor bar. For the experiment, the faulty bar was broken by drilling a hole into the aluminum part as shown in Fig. 7.

For the investigated machine $\text{ratioCageR} = 9$ was estimated from the geometry, and in the same way, without getting into details, it was assumed that $\text{ratioCageL} = 9$, too. With these parameters the rotor bar and end ring segment parameters $R_{b,\text{sym}}$, $L_{b\sigma,\text{sym}}$, $R_{e,\text{sym}}$ and $L_{e\sigma,\text{sym}}$ are computed according to (8)–(11) for a given rotor resistance R'_r and a leakage inductance $L'_{r\sigma}$. In the Modelica simulation the broken bar was considered by setting the faulty bar resistance with index 1

$$R_{b[1]} = 100R_{b,\text{sym}}. \quad (17)$$

This resistance increase causes the current through this bar to sufficiently vanish.

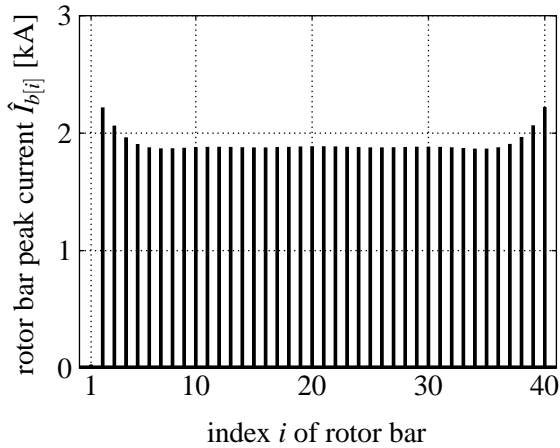


Figure 8: Peak values of the rotor bar currents; broken rotor bar with index 1; simulation results

7 Simulation Results

An electrical rotor asymmetry gives rise to a distortion of the current distribution of the rotor bars and end ring segments. The rotor bar currents can be computed from the rotor currents according to Fig. 3,

$$I_{b[i]} = I_{r[i]} - I_{r[i-1]}. \quad (18)$$

For nominal load and steady state operating conditions the peak values of the the sinusoidal currents of the rotor bars are depicted in Fig. 8. A time domain plot of some these currents (index 40, 1, 2 and 3) are shown in Fig. 9. The current of the broken rotor bar (index 1) is almost zero. Additionally, an interesting phenomenon can be observed. The currents of the directly adjacent rotor bars (e.g. index 40 and 2) are significantly larger than the currents of the remaining rotor bars. Due to this effect and the associated heat losses, the thermal stress of the directly adjacent rotor bars increases. This may also cause a damage of the adjacent bars which gives rise to an avalanche-like increase of the extend of the damage. Nevertheless, electrical rotor asymmetries spread relatively slow compared to other machine faults. A significant rise of the fault extend may thus happen within weeks, months or even years.

Due to the distortion of the current distribution in the rotor bars, the end ring current distribution changes, too. The rotor end ring currents of the A- and B-side can be defined by

$$I_{ea[i]} = I_{r[i]}, \quad (19)$$

$$I_{eb[i]} = I_{r[i]} + I_{eb}. \quad (20)$$

Without any asymmetry of either of the end rings, $I_{eb} = 0$ applies and therefore the currents of the A- and

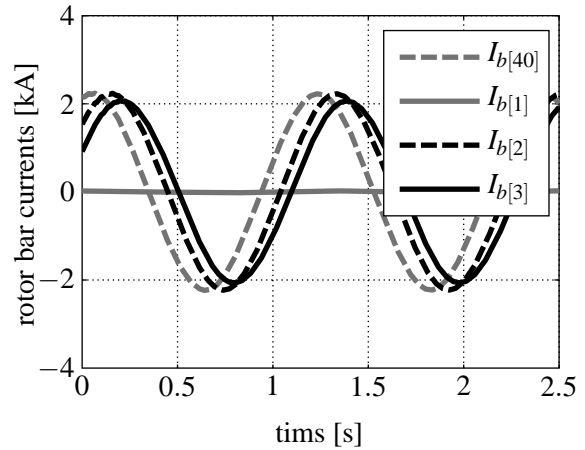


Figure 9: Rotor bar currents; broken rotor bar with index 1; simulation results

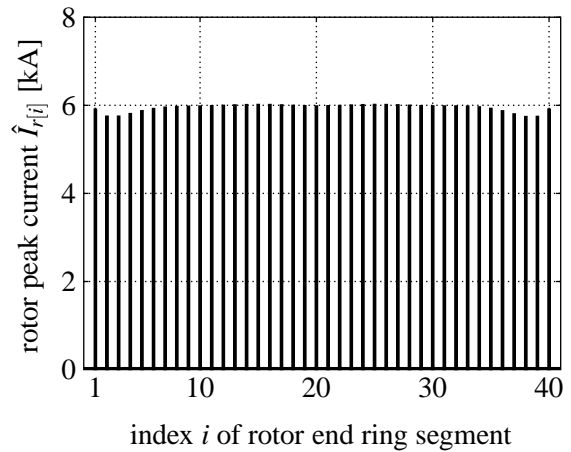


Figure 10: Peak values of the currents of the end ring segments; broken rotor bar with index 1; simulation results

B-side are equal. The peak values of the currents of the rotor end ring segments are depicted in Fig. 10.

The lower and upper side band harmonics of the current arise a few Hertz differing from the fundamental wave according to (13) and (15). For the investigated machine 50 Hz machine the Fourier spectrum of a stator current (phase 1) is depicted in Fig. 11. The lower and upper side band harmonics clearly arise at 48.6 Hz and 51.4 Hz. The magnitudes of these side band components are, however, much smaller than the magnitude of the fundamental. Therefore, electrical rotor asymmetries can usually not be determined from the time domain waveforms.

8 Measurement Results

The measured Fourier spectrum of on stator phase current is depicted in Fig. 12. Comparing this plot with

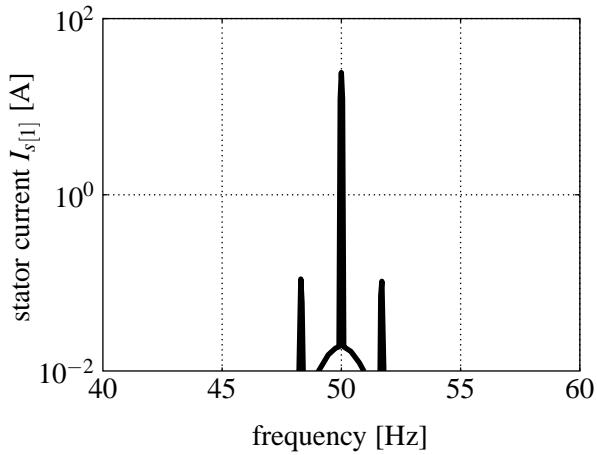


Figure 11: Fourier spectrum of the stator current $I_{s[1]}$; simulation results

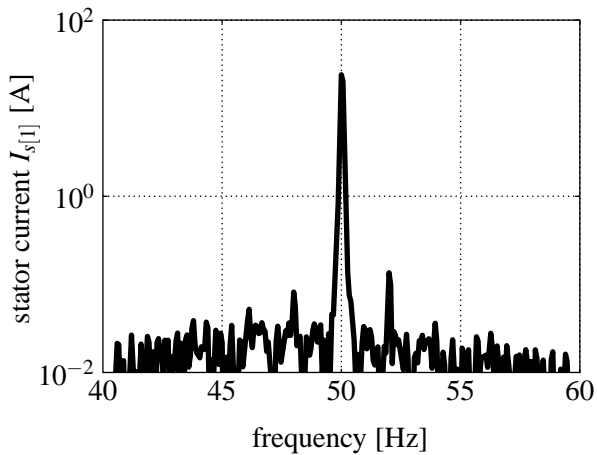


Figure 12: Fourier spectrum of the stator current $I_{s[1]}$; measurement results

the simulation results of Fig. 11 reveals, that the frequencies of the side band harmonics are the same, but the magnitudes show small deviations. The different magnitude of the side band harmonics is mainly related with the inertia of the drive which is not perfectly tuned such way that the magnitudes match. Additionally, due to the deviation of the modeled rotor bar and end ring resistance ratio from the real machine cage, some deviations in the simulation results may arise. With respect to the comparison of measurement and simulation results it should also be noted that in a real motor interbar currents can arise [9], which are not modeled in the ExtendedMachines library. These interbar currents lead to a current flow through the rotor teeth, first, in general, and, second, adjacent to the broken bar. Therefore, interbar currents may have an impact on the magnitudes of the side band currents.

9 Rotor Fault Detection Methods

Only severe rotor asymmetries can be detected through significant fluctuations of the amperemeters or wattmeters connected in the feeders of the machine. Upcoming electric rotor asymmetries require some more sophisticated detection methods. The most usual rotor fault detection methods are solely based on the measurement of one stator current. This class of methods is called *current signature analysis* (CSA) methods [10], [11], [12]. The measured current is then processed by either a fast Fourier transform [13] or a wavelet transform [14], [15] or some other signal processing techniques.

A second class of methods is based on *power signature analysis* (PSA), evaluating either total or phase power [7]. Additionally to CSA or PSA techniques, neural networks [16], [17] or Fuzzy based methods [18] may be applied.

A third class uses model based techniques for detecting a rotor fault. One model based technique is the Vienna Monitoring Method (VMM) which was introduced in 1997 [19]. This method evaluates two mathematical machine models and calculates torque for each model. In case of a fully symmetrical machine both models calculate the same torque and torque difference is zero. An electrical rotor asymmetry gives rise to side band currents and double slip frequency torque oscillations. These torque oscillations are differently sensed by the two mathematical models due to the different model structure. This leads to different magnitudes and phase shifts of the double slip frequency torque oscillation derived by the two models. Therefore, torque difference shows a double slip frequency oscillation which indicates an electrical rotor fault.

The torque difference, divided by the average load torque, serves as quantity to determine the fault indicator through a certain data clustering technique. The fault indicator is basically the magnitude of the double slip frequency oscillation of the torque difference, divided by the average load torque. The particular advantage of the VMM is that it provides a reliable fault indicator independent of load torque, speed, supply and inertia of the drive [20], [21], [22].

The VMM has been applied to both the simulation and measurement results. For the simulation results the determined fault indicator is 0.0093 and the VMM applied to the measured data leads to a fault indicator of 0.0105. This is a deviation of about 11%. Considering no parameter tuning of the simulation model this is a satisfactory result.

10 Conclusions

For squirrel cage induction machines the background of electrical rotor asymmetries is discussed. A rotor topology models for handling cage asymmetries is introduced and the implementation in the ExtendedMachines library is presented.

A 18.5 kW induction machine with one broken rotor bar out of 40 bars is investigated. The simulated and measured stator current Fourier spectrum are compared and shown good coherence. Additionally, the simulation and measurement data are applied to a model based rotor fault detection method – the Vienna Monitoring Method. The comparison of the two fault indicators determined by the Vienna Monitoring Method reveals a deviation of about 11%. Considering that no parameter tuning of the simulation model has been performed, this is a satisfactory matching result.

References

- [1] C. Kral, A. Haumer, and F. Pirker, “A modelica library for the simulation of electrical asymmetries in multiphase machines - the extended machines library,” *IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives, The 6th, SDEMPED 2007, Cracow, Poland, 2007*.
- [2] C. Kral, *Modellbildung und Betriebsverhalten einer Asynchronmaschine mit defektem Rotorstab im Läuferkäfig einschließlich Detektion durch die Vienna Monitoring Method*. PhD thesis, Technische Universität Wien, 1999.
- [3] G. Müller, *Elektrische Maschinen - Theorie rotierender elektrischer Maschinen*. Berlin: VEB Verlag Technik, 2 ed., 1967.
- [4] F. Filippetti, G. Franceschini, C. Tassoni, and P. Vas, “Impact of speed ripple on rotor fault diagnosis of induction machines,” *Proceedings of the International Conference on Electrical Machines, ICEM*, pp. 452–457, 1996.
- [5] M. Schagginger, *Diplomarbeit-Luftspaltfelduntersuchungen an umrichter gespeisten Asynchronmaschinen im Hinblick auf elektrische Unsymmetrien im Rotorkäfig*. Wien: TU, 1997.
- [6] H. Henao, C. Demian, and G.-A. Capolino, “A frequency-domain detection of stator winding faults in induction machines using an external flux sensor,” *Issue: 5 Industry Applications, IEEE Transactions on*, Volume: 39, pp. 1272–1279, 2003.
- [7] Z. Liu, X. Yin, Z. Zhang, D. Chen, and W. Chen, “Online rotor mixed fault diagnosis way based on spectrum analysis of instantaneous power in squirrel cage induction motors,” *IEEE Transactions on Energy Conversion*, vol. 19, pp. 485–490, Sept. 2004.
- [8] S. Legowski, A. S. Ula, and A. Trzynadlowski, “Instantaneous power as a medium for the signature analysis of induction motors,” *IEEE Transactions on Industry Applications*, vol. 32, no. 4, pp. 904–909, 1996.
- [9] R. Walliser, “The influence of interbar currents on the detection of broken rotor bars,” *ICEM*, pp. 1246–1250, 1992.
- [10] M. Benbouzid, “A review of induction motors signature analysis as a medium for fault detection,” *Annual Conference of the IEEE Industrial Electronics Society, IECON*, pp. 1950–1955, 1998.
- [11] A. Bellini, F. Filippetti, G. Franceschini, C. Tassoni, R. Passaglia, M. Saottini, G. Tontini, M. Giovannini, and A. Rossi, “On-field experience with online diagnosis of large induction motors cage failures using mcsa,” *IEEE Transactions on Industry Applications*, vol. 38, no. 4, pp. 1045–1053, 2002.
- [12] G. Didier, E. Ternisien, O. Caspary, and H. Razik, “Fault detection of broken rotor bars in induction motor using a global fault index,” *IEEE Transactions on Industry Applications*, vol. 42, pp. 79–88, Jan.-Feb. 2006.
- [13] I. M. Culbert and W. Rhodes, “Using current signature analysis technology to reliably detect cage winding defects in squirrel-cage induction motors,” *IEEE Transactions on Industry Applications*, vol. 43, pp. 422–428, March-april 2007.
- [14] H. Douglas, P. Pillay, and A. Ziarani, “Broken rotor bar detection in induction machines with transient operating speeds,” *IEEE Transactions on Energy Conversion*, vol. 20, pp. 135–141, Mar 2005.

- [15] J. Antonino-Daviu, M. Riera-Guasp, J. Folch, and M. Palomares, "Validation of a new method for the diagnosis of rotor bar failures via wavelet transform in industrial induction machines," *IEEE Transactions on Industry Applications*, vol. 42, pp. 990–996, July-August 2006.
- [16] B. Ayhan, M.-Y. Chow, and M.-H. Song, "Multiple discriminant analysis and neural-network-based monolith and partition fault-detection schemes for broken rotor bar in induction motors," *IEEE Transactions on Industrial Electronics*, vol. 53, pp. 1298–1308, June 2006.
- [17] W. W. Tan and H. Huo, "A generic neurofuzzy model-based approach for detecting faults in induction motors," *IEEE Transactions on Industrial Electronics*, vol. 52, pp. 1420–1427, Oct. 2005.
- [18] E. Ritchie, X. Deng, and T. Jokinen, "Diagnosis of rotor faults in squirrel cage induction motors using a fuzzy logic approach," *Proceedings of the International Conference on Electrical Machines, ICEM*, pp. 348–352, 1994.
- [19] R. Wieser, C. Kral, F. Pirker, and M. Schagginger, "On-line rotor cage monitoring of inverter fed induction machines, experimental results," *Conference Proceedings of the First International IEEE Symposium on Diagnostics of Electrical Machines, Power Electronics and Drives, SDEMPED*, pp. 15–22, 1997.
- [20] C. Kral, F. Pirker, and G. Pascoli, "Influence of load torque on rotor asymmetry effects in squirrel cage induction machines including detection by means of the Vienna Monitoring Method," *Conference Proceedings EPE*, 2001.
- [21] C. Kral, H. Kapeller, J. Gragger, F. Pirker, and G. Pascoli, "Detection of mechanical imbalances during transient torque operating conditions," *5th IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives, SDEMPED*, 2005.
- [22] C. Kral, F. Pirker, and G. Pascoli, "The impact of inertia on rotor faults effects - theoretical aspects of the Vienna Monitoring Method," *IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives, The 6th, SDEMPED 2007, Cracow, Poland*, 2007.

Modeling and Simulation of a Large Chipper Drive

Hansjörg Kapeller Anton Haumer Christian Kral Gert Pascoli Franz Pirker
arsenal research
Giefinggasse 2, 1210 Vienna, Austria
hansjoerg.kapeller@arsenal.ac.at

Abstract

This paper presents a simulation model for a large chipper drive used in a paper mill. If the chipper drive is a slip ring induction motor, several advantages arise from using a rheostat in the rotor circuit. This paper will investigate the impact of a rotor circuit rheostat with respect to starting behavior and heavy duty load impulses. Furthermore an alternative chipper drive with a speed controlled squirrel cage induction machine will be presented. Both drives are modeled in Modelica. Simulation results are compared and discussed.

Keywords: Chipper drive; slip ring induction motor; squirrel cage induction machine; speed control; load impulses; simulation model

1 Introduction

Chipper drives are used in paper mills for crushing trunks and making wood chips. The nominal power of a motor used for such applications ranges from several 100 kW up to 2 MW and even more. A chipper drive is usually not operated continuously, because load impulse-arise only if a trunk is shredded. After that, the motor is not loaded until the next trunk is processed. The heavy duty load impulses (double the nominal torque) give rise to large motor currents which cause large voltage drops across the mains impedance. To the regulations of the standards, and the actual configuration and structure of the voltage supply, certain voltage drops (flicker effects) may not be exceeded during impulse load or starting operation. Under some circumstances, the chipper drive even has to start with some remaining parts of a trunk loaded which is a heavy duty condition for the motor.

In the following, two possible chipper drive applications are presented. First, a slip ring motor with rheostat in the rotor circuit and, second, a speed controlled squirrel cage motor is investigated.

1.1 Slip Ring Motor with Additional Resistances

Both, starting the chipper drive and heavy duty operating, cause large currents if no measures are taken. For a motor not being supplied by an inverter, high starting currents arise from the low locked rotor impedance of the induction motor [1]. It is therefore often useful to use a slip ring motor instead of squirrel cage motor. With a slip ring motor an additional rheostat in the rotor circuit can be used to increase the impedance. This gives rise to reduced starting currents and improves the torque speed characteristic with the effect, that reactions of the load impulses on the motor currents and voltage sags are diminished, too.

The significant disadvantages of a slip ring motor with additional resistances in the rotor circuit is the deterioration of efficiency due to additional losses in the external rotor resistances and the high abrasion of the brushes, which gives rise to an increased deposit of brush dust in the motor. This brush dust subsequently increases the risk of isolation breakdown and causes higher costs of maintenance.

1.2 Low Voltage Inverter Supplied Squirrel Cage Motor with Speed Control

An alternative approach which gives rise to reduced starting and load peak currents is a speed controlled inverter drive with squirrel cage motor (Fig. 1).

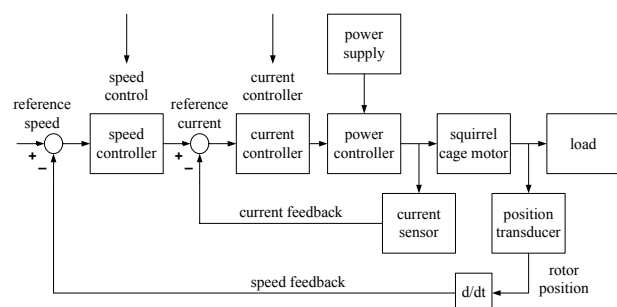


Figure 1: General diagram of a speed controlled drive

The basic topology of a speed controlled drive consists of the electric machine, the power converter, the power supply, cascaded current and speed controller, the mechanical load, the current sensor and the position transducer respectively [2]. The speed controller controls the reference stator current of the machine according to the deviation of the actual speed from the reference speed. The current controller has a build in limitation of the current to avoid overloading the machine. This leads to efficiency savings over a wide operating range and indicates an advantage compared to the realization with a slip ring motor. The higher investment costs due to the additional equipment – the entire speed control implementation with all necessary sensors – are disadvantageous, however.

1.3 Technical Data

Grid	
Frequency	50 Hz
RMS voltage, line-to-line	6000 V
Short-circuit apparent power	50 MVA
Short-circuit power factor	0.05
Transformer's nominal apparent power	1.8 MVA
Transformer's short-circuit p.u. voltage	0.06
Transformer's copper losses	17.5 kW

Table 1: Grid data

Chipper drive with slip ring motor	
Frequency	50 Hz
Number of pole pairs	2
RMS stator voltage, line-to-line	6000 V
RMS stator current	161.1 A
RMS rotor voltage, line-to-line	1500 V
RMS rotor current	595.3 A
Warm stator resistance per phase	129.0E-3 Ω
Stray stator inductance per phase	6.845E-3 H
Main inductance per phase	273.8E-3 H
Stray rotor inductance per phase	0.4631E-3 H
Warm rotor resistance per phase	8.729E-3 Ω
Motor rated power	1.5 MW
Motor rated rpm	1490.8 min^{-1}
Motor inertia	120 kg m^2
Load inertia	20000 kg m^2
Gear unit	1500:300 min^{-1}

Table 2: Data of the investigated chipper drive with slip ring motor

Chipper drive with squirrel cage motor	
Frequency	50 Hz
Number of pole pairs	2
RMS stator voltage, line-to-line	690 V
RMS stator current	1404.5 A
Warm stator resistance per phase	1.702E-3 Ω
Stray stator inductance per phase	0.10835E-3 H
Main inductance per phase	4.063E-3 H
Stray rotor inductance per phase	0.10835E-3 H
Warm rotor resistance per phase	1.135E-3 Ω
Motor rated power	1.5 MW
Motor rated rpm	1493.9 min^{-1}
Motor inertia	80 kg m^2
Load inertia	20000 kg m^2
Gear unit	1500:300 min^{-1}

Table 3: Data of the investigated chipper drive with low voltage inverter supplied squirrel cage motor

2 Simulation Models

For performing the Modelica simulations [3] the simulation tool Dymola is used. The behavior of the chipper drive – except for the inverter and control – can be modeled using the comprehensive *Modelica Standard Library* (MSL).

The free MSL provides a collection of standard components and component interfaces for many engineering domains. In the current version of the MSL all components for modeling the proposed chipper drive are offered. For the proposed simulation models mainly the *MultiPhase*, the *Machines* (includes, e.g., direct current, asynchronous induction and permanent magnet synchronous induction machines) and the *Rotational* packages of the MSL are used.

Since the drive controllers are not modeled in the MSL, controlled drives cannot be simulated with components of the MSL, only. Based on the *Machines* library [4] the *SmartElectricDrives* (SED) library [5] facilitates simulations of any electric drive application using different control structures and strategies.

The SED library contains models for the components used in a state-of-the-art electric drive. Sources (batteries and a PEM fuel cell), converters (ideal and power balanced), loads, process controllers, sensors, etc. are provided in this library. In the SED library, two classes of drives simulations are provided. The first class models quasi stationary drives, the second class uses the transient models of the MSL. For fast

simulations regarding energy consumption or the efficiency of a drive, the models of the *QuasiStationaryDrives* can be used. They have been modeled with the aim to neglect all electrical transients in the machines. Mechanical transients due to the rotor inertia are considered, however. A great benefit of the *QuasiStationaryDrives* is the remarkable shorter simulation time and the reduced number of input parameters due to simpler controller configuration and the neglect of switching effects. For the analysis of current spikes due to converter switching the *TransientDrives* have to be used. By choosing this lower level of abstraction the user pays the price of more computing time due to a high number of switching events caused by the inverter.

Besides all elementary components that give the user the freedom of building up an entire controlled machine, ‘ready to use’ models of drives are provided. These models can be used to conveniently and quickly arrange simulations [5]. The ‘ready to use’ models contain converters, measurement devices and a *field oriented control* (FOC).

2.1 Slip Ring Motor with Additional Resistances

A slip ring motor with an additional rheostat and an external constant resistance in the rotor circuit can be used to increase the impedance (Fig. 2). This gives rise to reduced starting currents and improves the torque speed characteristic (Fig. 3).

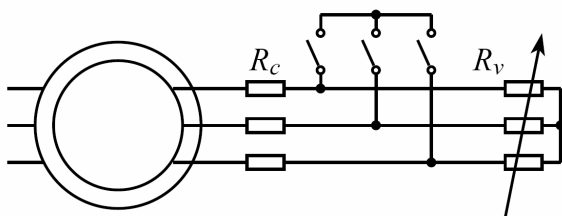


Figure 2: Three phase rheostat of a slip ring induction motor

The total rotor circuit resistance R_r^* consists of the actual rotor (winding) resistance R_r and the external resistance, which in turn consists of the variable resistance of the rheostat R_v and an external constant resistance R_c :

$$R_r^* = R_r + R_v + R_c. \quad (1)$$

During the start-up, the resistance of the rheostat R_v is reduced along a linear ramp. The duration of the ramp has to be chosen according to the actual inertia and starting conditions of the entire chipper drive. After reaching nominal speed, the variable resistance, R_v , is short circuited.

If the motor is not loaded, the resistance R_c does not have a significant influence on the motor current and speed. If the motor is loaded with a constant load torque the stationary speed depends on the actual resistance R_c according to

$$\frac{R_r}{s} = \frac{R_r + R_c}{s_c}, \quad (2)$$

where s denotes the slip for the case without external rotor circuit resistance, and s_c is the slip for the case with the external rotor circuit resistance R_c [6].

The total rotor circuit resistance leads to a scaled torque-slip characteristic. Therefore, load impulses can be covered partially by the stored energy of all rotating masses. The stationary torque speed characteristic of an induction motor is shown in Fig. 3, the stator current versus speed is shown in Fig. 4. For a short circuited slip ring rotor ($R_r^* = R_r$) the torque speed characteristic shows a very low starting torque and a starting current of approximately 5 times the nominal current. For $R_r^* = 21R_r$ the stationary characteristics show a significant improvement. In this case the locked rotor torque is close to the breakdown torque and the locked rotor current is less than 4.5 times the nominal current.

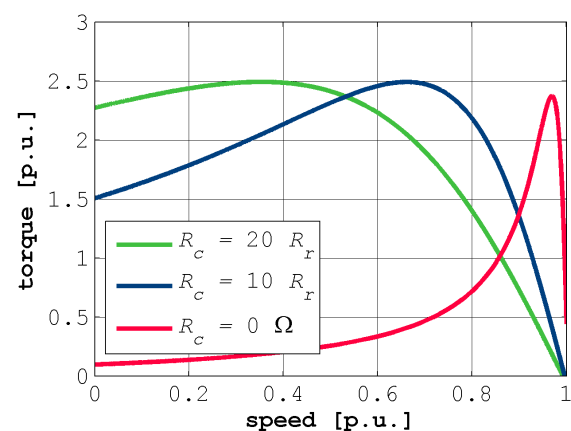


Figure 3: Stationary torque versus speed of a slip ring motor with external rotor resistance

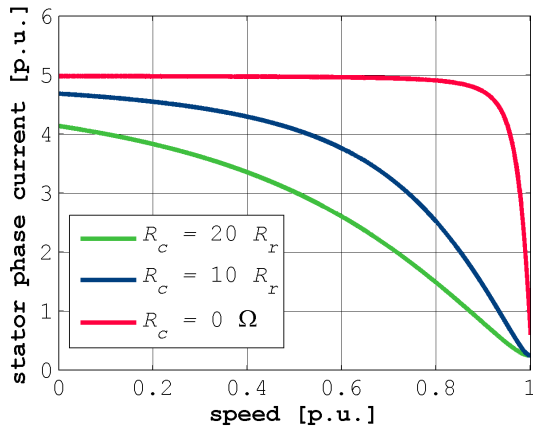


Figure 4: Stationary current versus speed characteristic of a slip ring motor with external rotor resistance

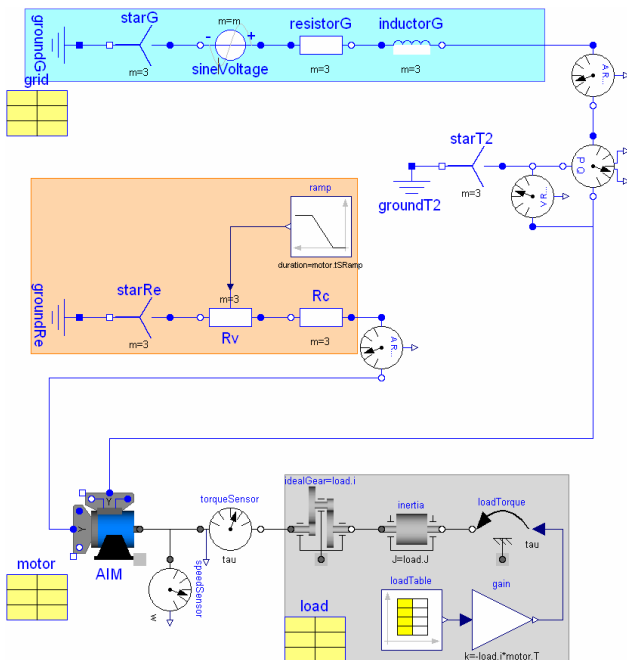


Figure 5: Chipper drive with slip ring motor

Figure 5 depicts the Modelica model of the chipper drive realized with a slip ring motor. The 6 kV / 50 Hz voltage supply is modeled by three sinusoidal supply voltages (sineVoltage) which are star (starG) connected. The overall mains impedance including all transmission lines and transformers, is modeled by a series connection of a three phase resistor (resistorG) and a three phase inductor (inductorG). For having the *root mean square* (RMS) values of the voltages and currents in the simulation results available, an RMS voltmeter and amperemeter are connected into the circuit. In addition to these instruments, a power sensor is used to measure the characteristic power terms of the circuit. The stator winding of the slip ring induction

motor (AIM) is star connected, the stator terminals are connected with the mains impedances, incorporating the instruments for voltage, current and power measurement. The rotor circuit of the slip ring rotor is also star connected. The slip ring terminals are series connected to a constant resistor (R_C) and a variable resistor (R_V). In the simulation model depicted in Fig. 5 the variable resistor is controlled by a ramp during the start-up of the motor. The signal inputs of the variable resistor, however, can be controlled by any other strategy as well. The mechanical shaft of the induction motor is connected with a torque and speed sensor. The power is transmitted through a gear (idealGear) to the load torque (loadTorque) model. The signal input of the load torque model is supplied by a time table (loadTable) modeling impulse loads.

2.2 Low Voltage Inverter Supplied Squirrel Cage Motor with Speed Control

In Fig. 6 a speed controlled chipper drive with a squirrel cage induction motor is presented. This drive uses components of the SED library. The voltage supply and measurement is the same as in the previous model, except that a transformer is used to provide 690 V to the low voltage drive.

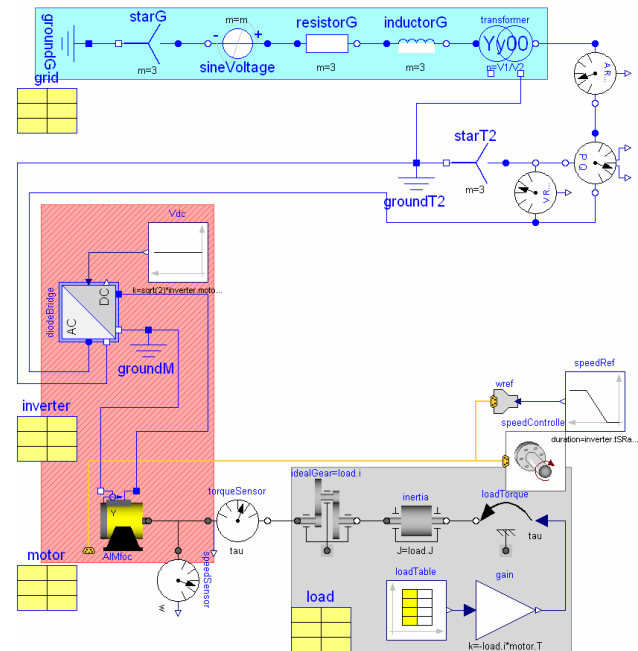


Figure 6: Modelica model of a speed controlled chipper drive with a squirrel cage motor

The transformed supply voltage (transformer) is rectified (diodeBridge) and provides the intermediate circuit voltage for the inverter. The idealized rectifier does not take into consideration the typical

non-sinusoidal waveform of a diode bridge. Therefore the supply current is rather comparable to that of an IGBT rectifier. The squirrel cage induction motor is supplied by a DC/AC-inverter which is implemented in the field oriented controlled *QuasiStationaryDrive* model (AIMfoc). The mechanical load model of the slip ring motor and the inverter drive are the same, however.

The components of the cascade control system anticipated in Fig. 1 can be parameterized separately [7]. Starting from the innermost to the outermost closed loop, various parameterization methods can be applied to achieve the desired dynamic behavior [8].

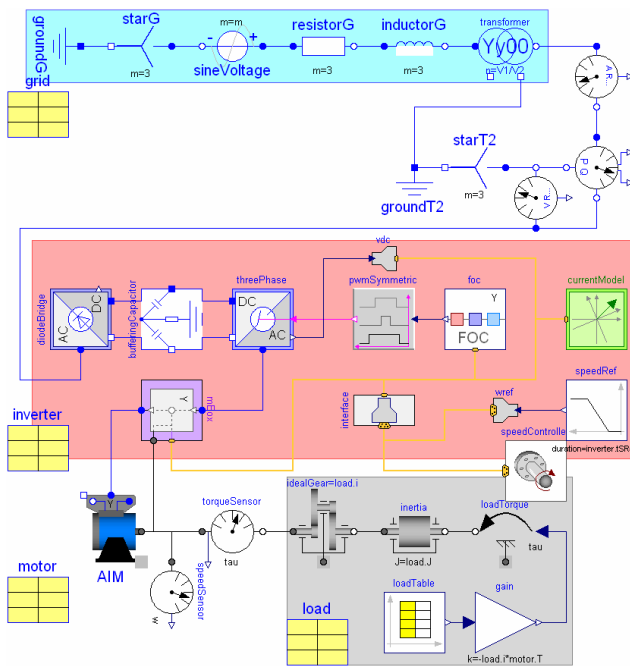


Figure 7: Full transient Modelica model of a speed controlled chipper drive with a squirrel cage motor

In Fig. 7 the full transient Modelica model of a speed controlled chipper drive with a squirrel cage induction motor is presented. This drive uses components of the SED library, again. The voltage supply and measurement is the same as in the quasi stationary model, but using an ideal switching diode rectifier. Additionally, the machine inverter is modeled in detail, not being integrated in the drive model. The detailed model of the ideal switching inverter leads to high number of events during simulation, and gives rise to significantly longer simulation time.

The dominant mechanical time constants are significantly greater than the electrical time constants. Provided that current peaks due to inverter switching may be neglected for this investigation, it is reasonable to use the *QuasiStationaryDrive* model which saves a significant amount of simulation time.

3 Simulation Results

In the presented results each load impulse has the same duration (2 s), equal rise and fall times (0.1 s) and the torque amplitude is twice the nominal torque. The first load impulse starts at $t = 30$ s, the second impulse starts at $t = 40$ s (cp. Fig. 8 and Fig. 12).

3.1 Slip Ring Motor

Simulation results for the chipper drive with slip ring motor are depicted in Fig. 8–11, where $R_c = 0 \Omega$ and $R_c = 10R_r$ were set consecutively. The duration of the linear ramp for decreasing the variable resistor ($R_{v,max} = 40R_r$) is 10 s. From Fig. 8 and 9 it can be deduced that the torque and current, respectively, get reduced due to the additional resistances in the rotor circuit. The supply current of the slip ring motor drive equals the stator phase current.

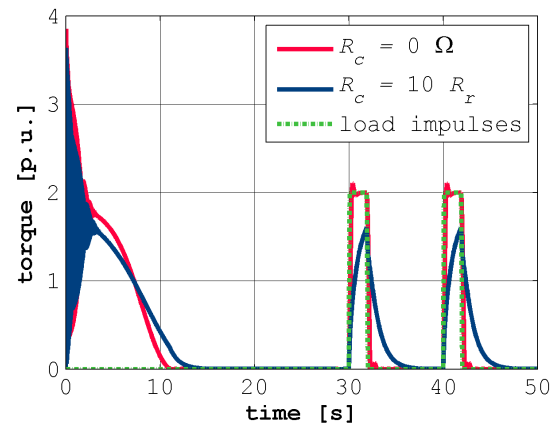


Figure 8: Load response of the slip ring motor ($R_c = 0 \Omega$ versus $R_c = 10R_r$ and $R_{v,max} = 40R_r$) and wave form of the modeled load impulses

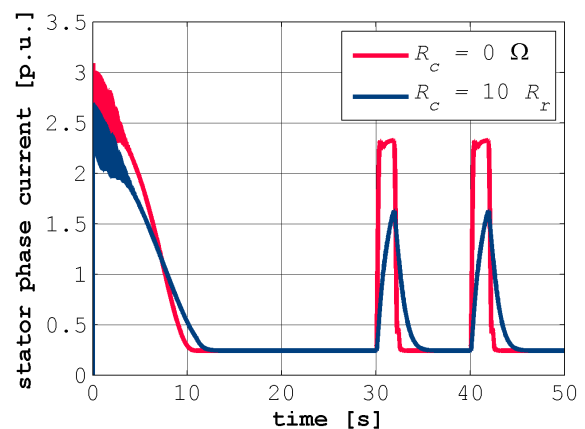


Figure 9: Stator phase current during start-up and during periodic loading of the chipper drive with slip ring motor ($R_c = 0 \Omega$ versus $R_c = 10R_r$ and $R_{v,max} = 40R_r$)

From Fig. 10 it is evident, that the additional resistance in the rotor circuit decreases the maximum voltage sag at the motor terminals. Figure 11 shows that higher external rotor resistance leads to larger speed drop during the load impulses.

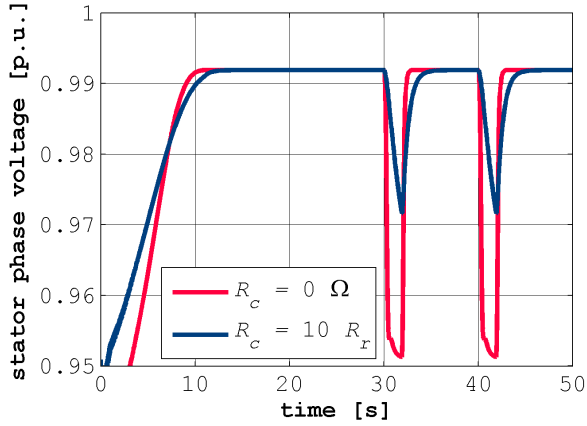


Figure 10: Stator phase voltage during start-up and during periodic loading of the chipper drive with slip ring motor ($R_c = 0 \Omega$ versus $R_c = 10R_r$ and $R_{v,max} = 40R_r$)

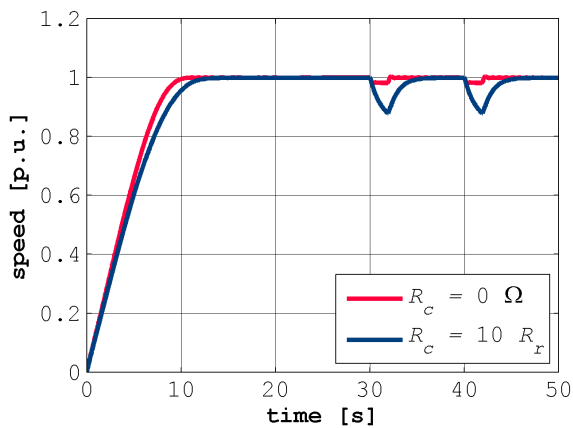


Figure 11: Speed during start-up and during periodic loading of the chipper drive with slip ring motor ($R_c = 0 \Omega$ versus $R_c = 10R_r$ and $R_{v,max} = 40R_r$)

3.2 Squirrel Cage Motor

Simulation results for the chipper drive with squirrel cage motor are depicted in Fig. 12–15. From Fig. 12 and 13 it can be seen, that the overloading of the machine is limited effectively by the current controller. The diminishing of the supply current peaks shows a significant improvement and the voltage drops are less than 2 % (cp. Fig. 14). Figure 15 illustrates, that the speed drop during the load impulses shows a similar dynamic characteristic as the chipper drive with slip ring motor ($R_c = 10R_r$ and $R_{v,max} = 40R_r$).

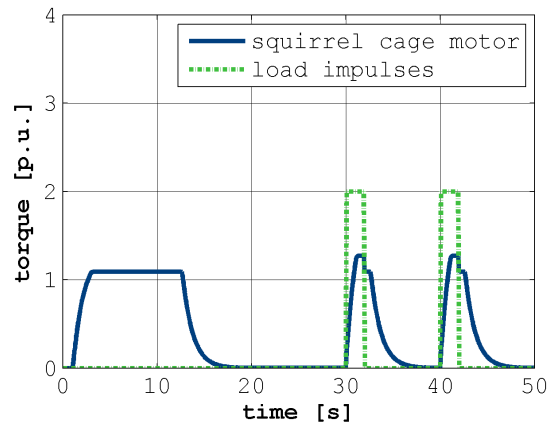


Figure 12: Load response of the speed controlled squirrel cage motor and wave form of the modeled load impulses

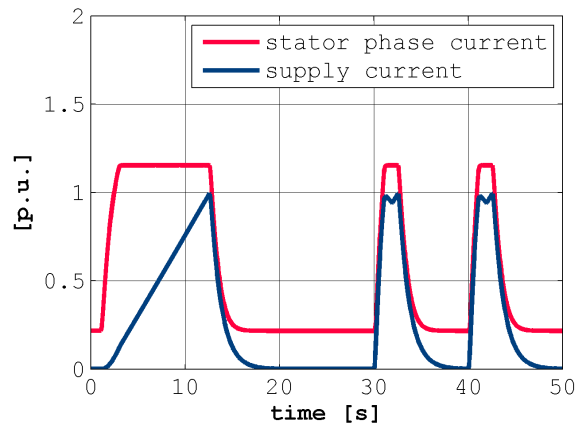


Figure 13: Supply current and stator phase current during start-up and during periodic loading of the chipper drive with speed controlled squirrel cage motor

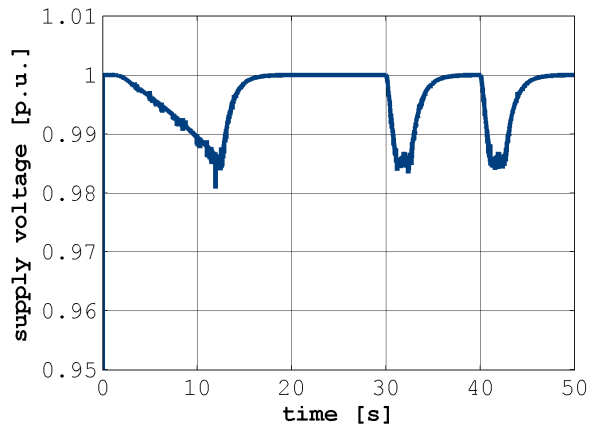


Figure 14: Supply voltage during start-up and during periodic loading of the chipper drive with speed controlled squirrel cage motor

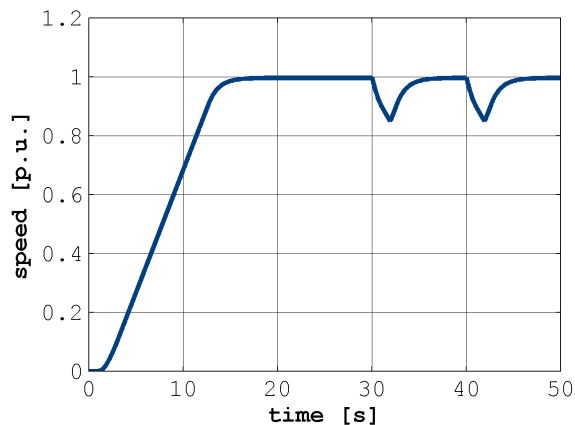


Figure 15: Speed during start-up and during periodic loading of the chipper drive with speed controlled squirrel cage motor

4 Conclusions

In this paper two large chipper drives are modeled in Modelica. The first drive is a slip ring motor with an external resistance in the rotor circuit; the second drive is a speed controlled squirrel cage motor. Both drives lead to a satisfactory reduction of voltage dips and current peaks during the start-up and the pulse load operation.

Using a chipper drive with a slip ring induction motor, the investment costs for the motor are higher than for an induction motor with squirrel cage. In case of the slip ring motor the rather simple additional equipment – the rheostat and its control – leads to low total investment costs.

Disadvantages of the slip ring motor are the deterioration of efficiency due to additional losses in the external rotor circuit and the high abrasion of the brushes, which gives rise to an increased deposit of brush dust in the motor. This brush dust subsequently increases the risk of isolation breakdown and causes higher costs of maintenance.

Contrarily, the inverter drive has lower costs of maintenance but higher investment costs. Although the squirrel cage induction motor is cheaper, more expensive additional equipment is needed: a transformer as well as a frequency inverter and the entire speed control unit with all necessary sensors.

However, the efficiency savings over a wide operating range and the absence of brushes indicates the main advantages compared to the slip ring motor.

References

- [1] P. L. Alger, *Induction Machines*. New York: Gordon and Breach Science Verlag, 1970.
- [2] S. Nasar and I. Boldea, *Electric Machines; Dynamics and Control*. 2000 Corporate Blvd, USA: CRC Press, 1st ed., 1993
- [3] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Piscataway, NJ: IEEE Press, 2004.
- [4] C. Kral and A. Haumer, *Modelica Libraries for Dc Machines, Three Phase and Poly-phase Machines*. Modelica Conference, 549-558, 2005.
- [5] H. Giuliani, C. Kral, J.V. Gragger and F. Pirker, *Modelica Simulation of Electric Drives for Vehicular Applications. The Smart Electric Drives Library*. ASIM, 2005.
- [6] R. Fischer, *Elektrische Maschinen*. München: C. Hanser Verlag, 5 ed., 1983.
- [7] O. Föllinger, *Regelungstechnik*. Heidelberg: Hüthig Verlag, 8th ed., 1994.
- [8] H. Lutz, W. Wendt, *Taschenbuch der Regelungstechnik*. Frankfurt am Main: Wissenschaftlicher Verlag Harri Deutsch, 5th enlarged ed., 2003.

Simulation and Validation of Power Losses in the Buck-Converter Model included in the SmartElectricDrives Library

Harald Giuliani Claus-J. Fenz Anton Haumer Hansjörg Kapeller
arsenal research
Giefinggasse 2, 1210 Vienna, Austria
harald.giuliani@arsenal.ac.at

1 Abstract

In this work a buck converter model included in the simulation software tool - the SmartElectricDrives library - is verified. The main focus is put on the converter losses. For these purposes a buck converter test bench was designed and set up. The power losses were measured according a defined series of measurements. Conduction and switching losses are investigated in this paper and their impact on the converter behavior is analyzed. As a result of the implemented losses concept the user should be able to parameterize the converter without comprehensive knowledge about transient transistor effects and data sheet availability.

2 Introduction

DC-DC converters are used to convert the unregulated DC input into a controlled DC output at a desired voltage level. The input voltage can be provided by a DC voltage source (e.g. a battery) or the DC-bus of an AC-DC-converter. The DC-DC converters are widely used in regulated switched mode DC power supplies and in DC motor drive applications.

In this paper one DC-DC-converter, like those utilized in electric vehicles, is investigated. The measurements on an electric vehicle emphasize the role of the dc-dc converter on the automotive market. In the investigated vehicle there are utilized three DC-DC converters in total. Two of them are used for feeding the electrically excited DC motor and one of them for charging the board system battery.

For shortening the period of development and reducing costs, simulation is a crucial step in the continuous design process. For the simulation of the energy flow of an entire hybrid vehicle [5] [6], the losses of each component have to be taken into account. So the modeling of the power losses in DC-

DC converter are relevant regarding the power balance of the whole system. Special software tools are necessary for this development process because the conventional simulation and calculation programs do not meet interdisciplinary and dynamic demands. In this contribution the Modelica [1] model of a DC-DC converter, taking the power dissipation into account, will be presented. Moreover the simulation results will be validated through measurements.

3 The Buck-Converter Model

3.1 The SmartElectricDrives Library

The SmartElectricDrives (SED) library [2] is written in Modelica and developed by arsenal research, with the focus on automotive applications. The SED library contains all basic machine types like asynchronous induction machines, permanent magnet synchronous machines, and direct current machines combined with various components needed for modern closed loop controlled drive systems like controllers and power electronic converters.

The most common DC-DC converters such as the chopper, the buck (step-down) converter, the boost (step-up) converter, the buck-boost converter and the full bridge are already included in the current version of the SED. The consideration of losses is planned to be implemented in the next release of the SED.

An important feature of the SED is that some components e.g. all the converter models are implemented at two different level of abstraction. The user can choose between power balance converters and ideal switching converters. In power balance converters the current flow is adjusted automatically due to the energy balance between the supply side and the load side considering switching and conduction losses. In switching converters the output voltage and the current flow is given by transistors switching states which are controlled by

pulse width modulation. Power balance converters are designed for simulations in which switching effects do not have to be considered. Their big advantage is that simulations work much faster with these models since the calculation effort for the power balance equation is much smaller compared to processing a large number of switching events.

3.2 The Buck-Converter

The basic structure of a buck converter is shown in Figure 1. A buck converter produces an average output voltage v_{Load} less than the DC input voltage v_{Supply} . By varying the duty ratio

$$D = \frac{t_{on}}{T_S} \tag{1}$$

of the switch, v_{Load} can be controlled.

t_{on} ...switch on duration

t_{off} ...switch off duration

T_S ...switching time period

L ...inductance

C ...capacitance

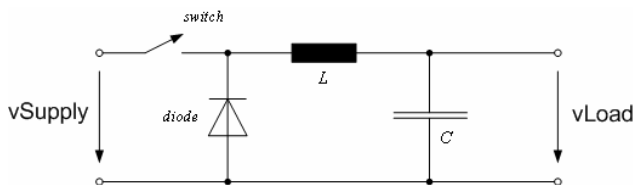


Figure 1: Basic structure of the buck-converter

The ideal output voltage v_{Load} of the buck converter without considering the conduction losses is:

$$v_{Load} = v_{Supply} \cdot D \tag{2}$$

Normally, the switch is either an IGBT (Insulated-Gate Bipolar Transistor) or a MOSFET (Metal Oxide Semiconductor Field Effect Transistor).

3.3 The Losses in the Buck-Converter Model

The losses of the buck converter [3] [4] are mainly conduction losses and switching losses. Conduction losses occur when the converter current flows through the internal power electronic components and involves a voltage drop, reducing the output voltage. Switching losses arise during the switching of the transistor or diode. During ‘switching on’ the voltage drop decreases whereas the current rises, causing high losses. Contrarily, during ‘switching off’ the losses are caused by a rising voltage drop and a decrease of the current.

The conduction and switching losses are considered in both the power balance and the ideal switching converter model.

- Conduction losses of the ideal switching model are affected by forward state-on resistance and the forward threshold voltage of the transistor and the diode, respectively. The power balanced converter model uses a controlled voltage drop to take the conduction losses into account. The losses of the inductor are considered too, whereas the losses in the capacitor are neglected. A parameter estimation function supports the user in determining consistent parameters.
- Both the power balance and the switching converter model use a controlled current sink at the input terminal to take the switching losses into account. For calculating the actual switching losses, the nominal switching power dissipation with respect to the rated operation point has to be known.

4 Calculation of Converter Losses

4.1 Conduction Losses

For the calculation of conduction losses it is assumed that the inductor current flows continuously. In this case one converter switching period consists of two converter circuit states (Figure 2 and Figure 3).

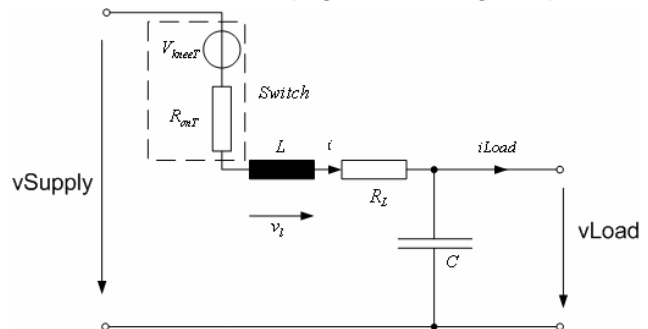


Figure 2: Buck-Converter circuit state: switch on

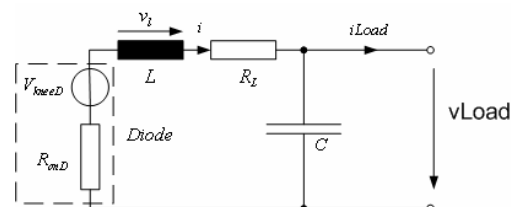


Figure 3 Buck-Converter circuit state: switch off

In steady state operation the waveform of voltages and currents repeat periodically. Therefore the integral of the inductor voltage v_L over one period, T_S , is zero:

$$\int_0^{T_s} v_l dt = \int_0^{t_{on}} v_l dt + \int_{t_{on}}^{t_{off}} v_l dt = 0 \quad (3)$$

According to the Kirchhoff's voltage laws applied to both converter circuit states (Figure 2 and Figure 3), (3) leads to:

$$\left[v_{Supply} - v_{Load} - V_{kneeT} - i \cdot (R_{onT} + R_L) \right] \cdot D = \left[v_{Load} + i \cdot (R_{onD} + R_L) + V_{kneeD} \right] \cdot (1 - D) \quad (4)$$

From (3) we obtain the average voltage drop between ideal (2) and real output voltage:

$$\Delta v = v_{Supply} \cdot D - v_{Load} = \left[V_{kneeT} + i \cdot (R_{onT} + R_L) \right] \cdot D + \left[V_{kneeD} + i \cdot (R_{onD} + R_L) \right] \cdot (1 - D) \quad (5)$$

The conduction losses of the power transistor

$$P_{C_T} = \left[i^2 \cdot (R_{onT} + R_L) + V_{kneeT} \cdot i \right] \cdot D \quad (6)$$

and the diode

$$P_{C_D} = \left[i^2 \cdot (R_{onD} + R_L) + V_{kneeD} \cdot i \right] \cdot (1 - D) \quad (7)$$

sum up to the total conduction losses:

$$P_C = P_{C_T} + P_{C_D} \quad (8)$$

Equation (6) and (7) prove that the model of the ideal switching power semiconductors inherently model the conduction losses. The voltage drop of the power balance model is based on (5).

v_l ... inductor voltage

v_{Supply} , v_{Load} ... average supply voltage, average load voltage

R_{onT} , R_{onD} ... state-on resistance of transistor/ diode

R_L ... inductor resistance

V_{kneeT} , V_{kneeD} ... forward threshold voltage of transistor/diode

i ... average inductor current (equals average load current)

i_{Load} ... average load current

4.2 Switching Losses

Detailed modeling of the switching losses through switching events leads to a high numeric effort. Therefore the average of these losses according to

(9)-(15) is taken into account in the SED buck-converter.

$$P_S = P_{S_T} + P_{S_D} \quad (9)$$

$$P_{S_T} = P_S \cdot (1 - r_{S_D}) \cdot \frac{i_T}{i_{T_Nom}} \cdot \frac{v_{blocking_T}}{v_{blocking_T_Nom}} \cdot \frac{f}{f_{Nom}} \quad (10)$$

$$P_{S_D} = P_S \cdot r_{S_D} \cdot \frac{i_D}{i_{D_Nom}} \cdot \frac{v_{blocking_D}}{v_{blocking_D_Nom}} \cdot \frac{f}{f_{Nom}} \quad (11)$$

The blocking voltages of the transistor and the diode are:

$$v_{blocking_T} = v_{Supply} + i_D \cdot R_{onD} + V_{kneeD} \quad (12)$$

$$v_{blocking_D} = v_{Supply} - i_T \cdot R_{onT} - V_{kneeT} \quad (13)$$

The nominal blocking voltages of the transistor and the diode are:

$$v_{blocking_T_Nom} = VDC + i_{T_Nom} \cdot R_{onD} + V_{kneeD} \quad (14)$$

$$v_{blocking_D_Nom} = VDC - i_{D_Nom} \cdot R_{onT} - V_{kneeT} \quad (15)$$

P_{S_T} , P_{S_D} ... switching losses in transistor/ diode

P_S ... sum of switching losses

r_{S_D} ... ratio of switching losses in the diode

i_T , i_{T_Nom} ... transistor current/ nominal transistor current

i_D , i_{D_Nom} ... diode current, nominal diode current

$v_{reverse_T}$ / $v_{reverse_D}$... blocking voltage of transistor/diode

$v_{reverse_T_Nom}$ / $v_{reverse_D_Nom}$... nominal blocking voltage of transistor/ diode

VDC ... nominal DC supply voltage

f , f_{Nom} ... switching frequency/ nominal switching frequency

5 Measurement Setup

The measurement setup is shown in Figure 4.

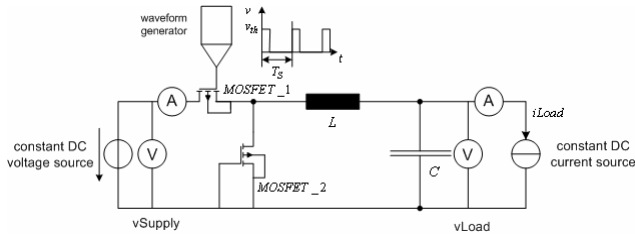


Figure 4 Measurement setup

Two power MOSFETs are used. One of them (MOSFET_1) is used as switch. The freewheeling diode of the other one (MOSFET_2) is used as buck diode. The converter is fed by a constant DC-voltage. A constant current source is used as the converter load. The switching MOSFET_1 is controlled by a waveform generator. The pulsewidth of the waveform generator is variable from 0 to 1 (0 means an open switch, whereas at duty cycle 1 the switch is closed all the period).

The parameters in Table 1 are obtained from the data sheet [7] of the used MOSFETs and measurements, respectively.

MOSFETs	$R_{onT} = 7m\Omega$ (data sheet)
	$R_{onD} = 3m\Omega$ (data sheet)
	$V_{kneeT} = 0V$ (data sheet)
	$V_{kneeD} = 0.8V$ (data sheet)
Inductor	$R_L = 2.9m\Omega$ (measured)
	$L = 4.57\mu H$ (measured)
Capacitor	$C = 1000\mu F$ (measured)

Table 1

The conducted measurements are summarized in the Table 2.

ID	V_{DC}	i_{Load}	f	duty cycle
M1	30 V	5 A	100kHz	0.2,..0.8 (step 0.1)
M2	30 V	10 A	100kHz	0.2,..0.8 (step 0.1)
M3	30 V	15 A	100kHz	0.2,..0.8 (step 0.1)
M4	30 V	20 A	100kHz	0.2,..0.8 (step 0.1)
M5	30 V	25 A	100kHz	0.2,..0.8 (step 0.1)
M6	30 V	30 A	100kHz	0.2,..0.8 (step 0.1)
M7	30 V	35 A	100kHz	0.2,..0.8 (step 0.1)
M8	30 V	40 A	100kHz	0.2,..0.8 (step 0.1)

Table 2

Figure 5 illustrates the obtained power losses versus duty cycle for measurement M5.

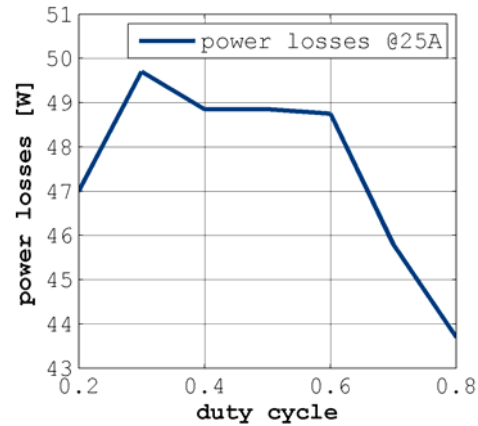


Figure 5: Power losses @25A

6 Simulation and Comparison with Measurement Results

6.1 Simulation

Figure 6 shows the simulation model of the buck converter. The operation conditions summarized in Table 1 are also applied to the simulations.

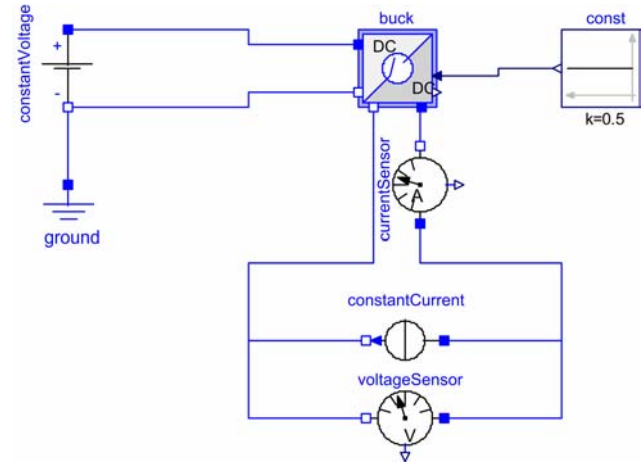


Figure 6: Simulation model of the buck converter

The measurement result of the total power losses at the nominal operating point (load current of 25 A; duty cycle of 0.5; switching frequency 100 kHz) is $P_l = 48.85W$. The conduction losses at the nominal operating point are calculated according to (6)-(8) $P_C = 14.93W$. The switching losses are calculated as the difference between total losses and conduction losses: $P_S = 33.92W$. The simulation reference values of switching losses in the nominal operating are defined by this calculated value. The simulation is fed by this value of switching losses. By changing the nominal operating point the value of the power dissipation is calculated by equations (9)-(15).

In Figure 7 the measured and the simulated power losses versus the duty cycle for measurements M2, M5 and M8 are compared.

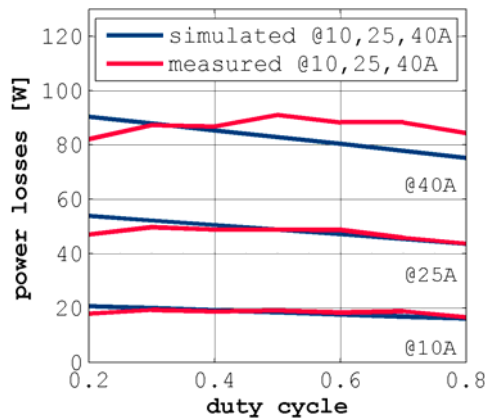


Figure 7: Comparison of measured and simulated power losses at different load currents

The output voltage is strongly dependent on the duty cycle, and almost independent of the load current. Figure 8 presents the load voltage versus duty cycle at three different load currents.

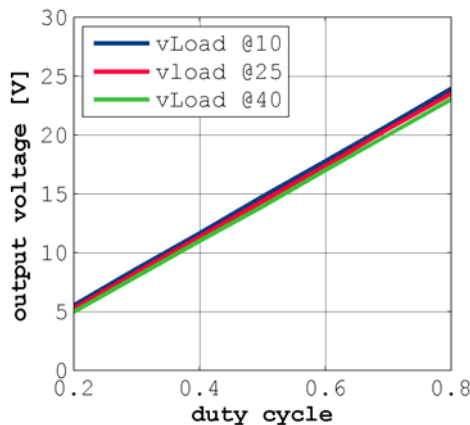


Figure 8: Measured output voltage at different load currents

The deviation of the simulated output voltage from the measurements results is less than 4 % (Figure 9).



Figure 9: Simulated voltage error at different duty cycles and load currents

6.2 Parameter Optimization

Temperature dependency and other physical effects lead to simulation results deviating from measurements. The measured losses in Figure 10 were linearly approximated, leading to the fitted parameters shown in Table 3.

MOSFETs	$R_{onT} = 13.8m\Omega$
	$R_{onD} = 3m\Omega$
	$V_{kneeT} = 0V$
	$V_{kneeD} = 0.5245V$

Table 3

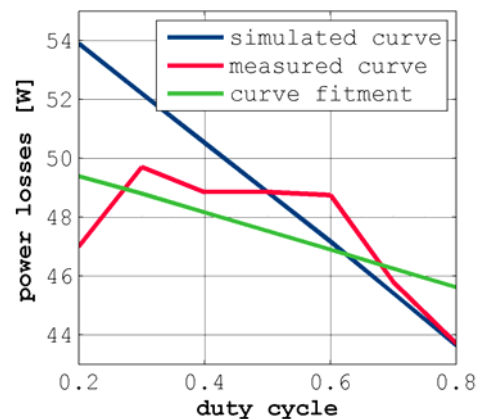


Figure 10: Power losses curve fitment

The change of the operation condition summarized in Table 1 is applied to the simulations.

Figure 11 illustrates the average errors (average error using parameter set from Table 1 and average error using parameter set from Table 3) of the simulated

losses from the measurement results at different load currents. Average error means the averaged error value over the duty cycle at a specific load current.



Figure 11: Average errors

It is obvious that the simulation with the new parameter set delivers better results in a large range.

7 Conclusion

In many applications DC-DC power converters are employed in a variety of applications, including power supplies for computers, power systems and telecommunications equipment, as well as dc motor drives.

For the simulation of the energy flow of an entire hybrid vehicle, the losses of each component have to be taken into account. The consideration of losses in DC-DC converter simulations should be organized user-friendly. This means that without big knowledge of all converter elements parameter it should be possible to carry out significant simulation results for a large operating range. As the measuring and simulating results have already shown, this target is fulfilled by the in the SED implemented DC-DC buck converter. The conduction losses are defined by forward resistances and threshold voltages. These parameters can get by data sheets or by measurements. To consider the switching losses, the nominal switching power dissipation with respect to the rated operation point has to be known. An optimization in a sub-operating range is easy done by calculating new parameters from the linear approximated measured power losses curve and using this improved set of parameters in the simulation.

References

- [1] P. Fritzon, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Piscataway, NJ: IEEE Press, 2004.
- [2] J.V. Gragger, H. Giuliani, The SmartElectricDrives Library -- Powerful Models for Fast Simulations of Electric Drives, 5th International Modelica Conference 2006, Vienna, Austria, 2006
- [3] N. Mohan, T.M. Undeland, W. Robbins, Power Electronics, J. Wiley Verlag, 1989
- [4] B. Bose, Power Electronics and Motor Drives, Elsevier, 2006
- [5] H. Giuliani, D. Simic, J.Gragger, C. Kral, C, Optimization of a four wheel drive hybrid vehicle by means of the SmartElectricDrives and the SmartPowerTrains library, The 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium & Exposition, EVS22, 2006
- [6] D. Simic, H. Giuliani, C. Kral, J.V. Gragger, Simulation of Hybrid Electric Vehicle, 5th International Modelica Conference, Vienna, Austria, 2006
- [7] Data sheet, International Rectifier, IRFPS3810, PD-93912B, HEXFET® Power MOSFET

Real-Time Modelica Simulation on a Suse Linux Enterprise Real Time PC

Arno Ebner

Martin Ganchev

Helmut Oberguggenberger

Franz Pirker

Arsenal Research

Giefinggasse 2 – 1210 Vienna Austria

arno.ebner@arsenal.ac.at

Abstract

This paper presents a real-time simulation system for a Suse Linux Enterprise Real-Time (SLERT) operating system workstation. With this system can be executed Hardware-in-the-loop (HIL) simulation. HIL is the integration of real components and system models in a common simulation environment. The main focuses of this simulation system presented in this paper are the development and the validation of simulation models of electric components (e.g. battery systems, electric drives etc.) for Hybrid (HEV) and Electric Vehicles.

The system is based on a Linux real-time computer with numerous analogue/digital input and output channels and all simulation models are implemented in Modelica using the for example the SmartPowertrains and SmartElectricDrives libraries. The models are simulated with the Dymola simulation environment for Linux.

Keywords: Real-time simulation, hardware-in-the-loop simulation, hybrid electric vehicle

1 Introduction

Hardware-in-the loop (HIL) is a useful method for the testing hybrid and electric vehicles components and is important for the validation and the verification of implemented simulation models. The integration of real components and virtual models in a common simulation environment highly supports the development processes of hybrid and electric vehicle components like electric drive systems or energy storage systems.

At Arsenal Research vehicle models are implemented in Modelica using the powerful SmartPowerTrains and SmartElectricDrives libraries. These vehicle models can be simulated with the Dymola simulation environment. The proposed hardware-in-the-loop simulation workstation is based on a Dual

CPU Xeon system with a SUSE Linux Enterprise Real-Time (SLERT) operating system. This operating system allows CPUs to be shielded from other processes and guarantees a highly deterministic execution environment. This system guarantee short response times and fast cycle times which are essential to meet the requirements of HIL simulations of for example electric drive trains. On this computer system can be executed standard Linux application such as the Dymola simulation tool. With the Modelica Real-Time Interface software, which was developed by Arsenal Research an interconnection between the simulation tool and I/O processes was realized. On this system sampling times up to 400 microseconds can be achieved depending on the complexity of the simulation model and the number of input and output channels. The I/O functionality of the HIL workstation is realized by data acquisition cards on which can be connected the high-power energy storage test bench and the high dynamic dynamometer. The energy storage test bench has a maximum charge power of 48kW and a maximum discharge power of 44kW at a maximum voltage of 600V. With the drive train test bed (dynamometer) can be tested electric drives with a maximum power of 110kW and up to a speed of 8000rpm. To connect Electronic Control Units (ECU) to the HIL simulator there exists also a CAN interface to the simulation environment. In that way the simulation can for example generate reference values which will be transmitted by CAN to the testing system or for example the simulation receives signals from the ECU and can evaluate these signals.

Detailed information about the implemented hybrid and electric vehicle models and especially about their real-time capability will be given in the paper. Also the connection between the Dymola simulation tool and the I/O functionalities will be described. Finally as an application example for the test facility a HIL simulation of a real electric drive system (electric drive, power electronics and battery) for a two wheels vehicle will be shown.

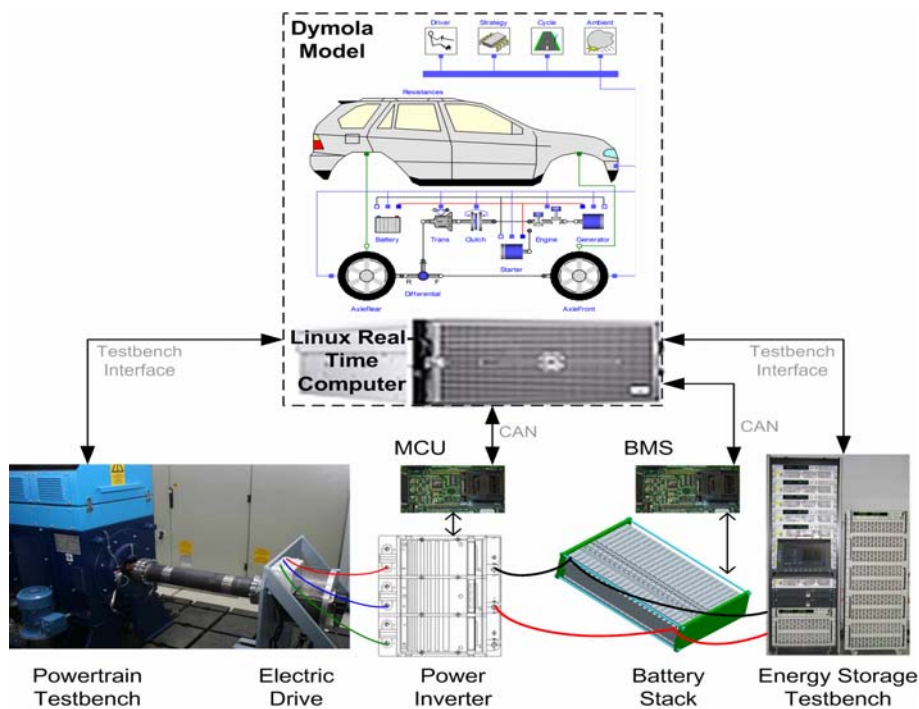


Figure 1: System overview of a HIL configuration for testing the electric components of a hybrid electric vehicle

2 System Overview

2.1 Real Time Computer

The main part of the proposed system is a 3.2 GHz Dual CPU Workstation with a SUSE Linux Enterprise Real-Time (SLERT) operating system. On this system operates a normal Dymola simulation environment for Linux on which can be implemented and simulated Modelica models. The effort of this system is that the simulation models can be executed in real-time directly from the simulation environment.

The proposed simulation computer with the SLERT operating system guarantees short response times and fast cycle times; such fast sampling time is for example required for HIL simulations of drive trains. A sampling frequency of over 2 kHz can be achieved depending on the complexity of the simulation model and the number of input and output channels. The input and output functionality to the test benches is provided by an analog/digital data acquisition card connected and for the digital communication with Electronic Control Units (ECU) by an external CAN interface.

2.2 Testing Infrastructure

The simulation environment on the Linux computer generates reference values for the drive train test

bench on which is connected the testing drive and for the energy storage test bench on which is for example connected a real battery. As feedback the simulation tool receives measured values from the real components. In figure 1 is given a possible schematic overview of the HIL system for testing the electric components for a hybrid or an electric vehicle. In this case the motor control unit (MCU), the power electronics (inverter), the electric drive, the battery management system (BMS) and the battery stack are the systems under test. The behavior of the vehicle for example the internal combusting engine, the driving dynamics, the ambient conditions, the drive cycle and the driver will be simulated on the real-time simulation workstation.

Table 1: Technical data of drive-train test-bench (dynamometer)

Peak Power	110 kW
Peak Torque	500 Nm
Peak Speed	8000 rpm
Control Modes	Torque or Speed
Ext. Data I/O Freq.	500 Hz

Table 2: Technical data of electric energy storage test-bench

Charge mode	
Peak Voltage	480 V
Peak Current	100 A @ 480 V or 500 A @ 60 V
Discharge mode	
Peak Voltage	600 V
Peak Current	540 A @ 600 V or 750 A @ 60 V
Control Modes	Voltage, Current, Power, Resistance

3 Modelica Real-Time Interface

The Modelica Real-Time Interface provides an interconnection between the Dymola simulation tool and the input and output functionalities of the system. The interface guarantees the synchronisation between the simulation time and the real time. Simulation time means the virtual time inside of the simulation time which will be matched with the time in the real world by the Real-Time Interface. The interface provides also the signal conditioning of the simulation variables to input and output values of the DAC channels, in that way the test benches can be connected directly to the simulation workstation.

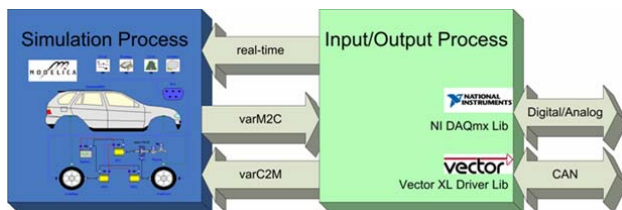


Figure 3: The two main processes for the real-time simulation

For a running real-time simulation two main processes the so-called Dymosim process and the RTmod process works together. The model and solver are executed in the Dymosim process, this is the left-handed process in figure 3. The model can be build with the RTInterface blocks for example for Trigger functions and data I/O. Data I/O is implemented by a Shred Memory and synchronized by semaphores. All functionality in Modelica is implemented using external c-functions and Modelica standard blocks. The RTmod process (right-handed process in figure 3) is completely c-written it includes the Frequency Based Scheduler synchronization the Data I/O over shared

memory and the DAQ functions for analoge and digital In- and Output.

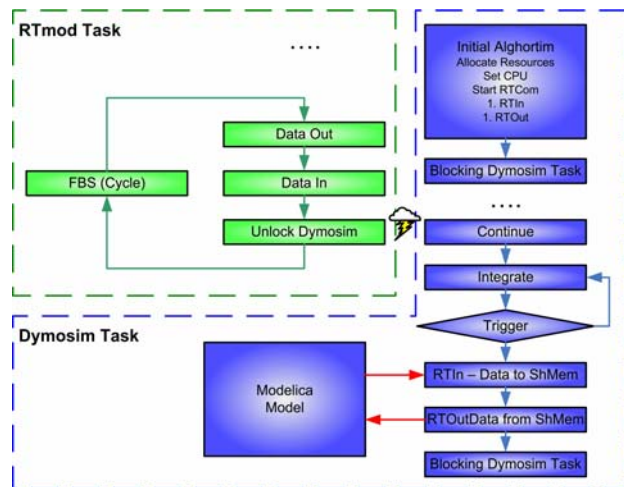


Figure 2: Functional overview of the Modelica real-time interface

4 Application Examples

4.1 Hybrid Electric Vehicle HIL Simulation

A hardware-in-the-loop simulation of a HEV system will be shown as an application example. The behavior of a Mild HEV is simulated on the Linux real-time computer on which is connected a real electric drive with a peak power of 12 kW and the motor control unit for this electric drive. The whole HEV is modeled in Modelica using the SmartPowertrains library and the SmartElectricDrives library. Based on suited power requirements pointed out by a drive cycle, the simulation model generates the reference values for the testing drive, (e.g. a reference torque) which are communicated to the Motor Control Unit (MCU) via the CAN bus and the power inverter generates the electric signals for the drive. The measured torque in the load cell of the test bench corresponds to the torque of the power train, this feedback value goes back the simulation model and new reference values can be calculated.

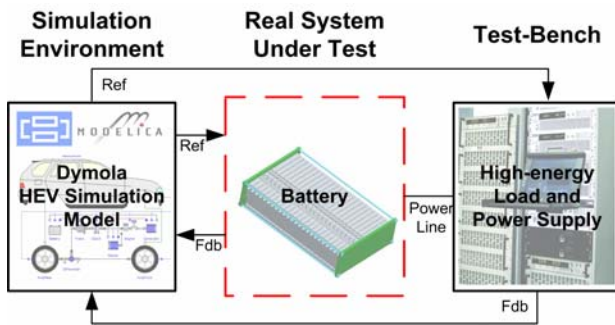


Figure 4: HEV hardware-in-the-loop simulation

4.2 Electric Two-Wheeler HIL Simulation

The capabilities of electric two-wheelers are intensively investigated due to their high potential for improved urban mobility as well as tourist or sport application. Nevertheless existing products are in many cases lacking of appropriate range or power to achieve a broad (satisfactory) customer acceptance. To evaluate the technological potential of electric two-wheelers different e-scooter concepts have been analyzed by simulation. Based on the simulation environment available at arsenal research an electric scooter model was set up and simulations using different electric drive configurations have been performed. The results allow for determination of high power scooter concepts. For validation and demonstration of a high power two-wheeler an e-scooter prototype has been realized and tested. The simulation and test results in e.g. performance and range will be given as well as the further improvement potential will be discussed.

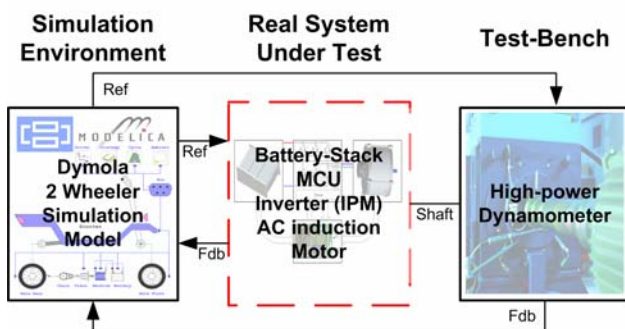


Figure 5: Electric two-wheeler hardware-in-the-loop simulation

5 Conclusion

In the provided paper is presented a HIL simulation solution for development and for testing of components for HEV's. The powerful hardware of the simulation computer and the real-time operating system allows the simulation of complex models with a fast data exchange with the real components which are connected to test benches. With the described *Modelica Real-Time Interface* it is possible to execute the HIL simulation direct from the *Dymola* simulation tool with which was implemented the model.

The benefit of this system is that a simulation engineer can for example validate an implemented model directly, matching the simulation results with the measured values at the connected real component. Or in the advanced development an engineer can test a developed component also when the entire system is not available; the prototype of the system exists only as a virtual model in the simulation.

References

- [1] Peter Fritzson, "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1", IEEE Press, Piscataway, NJ, 2004
- [2] M. Otter, H. Elmqvist, "Modelica – Language, Libraries, Tools, Workshop and EU-Project RealSim", Simulation News Europe, pp. 3-8, Dec. 2000
- [3] A. Ebner, A. Haumer, D. Simic, F. Pirker: "Interacting Modelica using a Named Pipe for Hardware-in-the-loop Simulation", 5th International Modelica Conference Proceedings Vol.1, pp. 261-266, Vienna, Austria, 2006.
- [4] A. Ebner, F. V. Conte, F. Pirker: "Rapid Validation of Battery Management System with a Dymola Hardware-in-the-Loop Simulation Energy Storage Test Bench", 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium & Exposition; pp. 1570-1574, Yokohama, Japan, 2006.
- [5] D. Simic, H. Giuliani, C. Kral, J. Gragger, "Simulation of Hybrid Electric Vehicles", 5th International Modelica Conference Proceedings Vol.1, pp. 25-32, Vienna, Austria, 2006.
- [6] J. Gragger, D. Simic, C. Kral, H. Giuliani, V. Conte, F. Pirker, "A Simulation Tool for

Electric Auxiliary Drives in HEVs - the Smart Electric Drives Library”, FISITA World Automotive Congress, Yokohama, Japan, 2006.

- [7] M. Noll, D. Simic, A. Ebner, “Evaluation of the Technological Potential of Electric Two-Wheelers Based on a High-Power Electric Scooter”, EET-2007 European Ele-Drive Conference, <http://www.ele-drive.com>, accessed on 2007-03-06

