Proceedings
of the 3rd International Modelica Conference,
Linköping, November 3-4, 2003,
Peter Fritzson (editor)

Jörgen Svensson and Per Karlsson
Dept. of Industrial Electrical Engineering and Automation, Lund University:
Adaptive signal management
pp. 179-188

Paper presented at the 3rd International Modelica Conference, November 3-4, 2003, Linköpings Universitet, Linköping, Sweden, organized by The Modelica Association and Institutionen för datavetenskap, Linköpings universitet

All papers of this conference can be downloaded from
http://www.Modelica.org/Conference2003/papers.shtml

Adaptive signal management

- A Modelica and C++ interaction example, the SignalFlow Library

Jörgen Svensson and Per Karlsson

Lund University
Dep. of Industrial Electrical Engineering and Automation
Box 118
SE-221 00 Lund
Sweden
jorgen.svensson@iea.lth.se

Abstract

This paper presents an adaptive signal management Modelica library, "SignalFlow", interconnected with a C++ class library. The objective is to simplify the signal exchange in large simulation models based on modular designs, which should correspond to the signal flow for real applications by representing common networks as models with general interfaces. The library enables automatic configurations during simulation using dynamic vectors and has additionally functions for exchanging several types of signals in both continuous and discrete mode. The work is an outcome for enabling "plug and produce" capabilities in scalable distributed power system applications that is exemplified.

1. Introduction

Communication interfaces are used in almost every technical application that needs signals to be exchanged. Control units may be embedded in components of varying sizes, where a component itself might be aggregated of others according to design and structure. Dependent on complexity, there are several signal levels both for horizontal and vertical interconnections [1,2]. This is complicated in real systems but even more in simulation environments (SE). Several SEs have hierarchical possibilities in modeling and define terminal types for signal exchange. However, numerous possibilities easy become a trap when using a multilevel hierarchy of signal interconnections. In Figure 1a it is shown that several different IO terminals easy become disordered as the system become larger. Different terminals represent various groups of signals that need to be used for connecting the models. It is easy to complicate the model structure by extending the number of terminal types, which cause many and tricky connections. For example, if adding a new type of terminal in model M12 in Figure 1a, each model at all levels need to be reconstructed by adding new terminals. If modeling a large model with many levels of aggregations it becomes even more complex. This is simplified by using a model representing a general communication bus, as shown in Figure 1b, where every instance is interconnected to the same bus independent in information level if so desired. Dependent on the number and types of communication interfaces there are alternative configuration opportunities for the signal exchange in the models. For example, if throughout using the same interface it might be practical for the user to be spared assigning identities to every single communication node. One model solution for the Figure 1b case requests a vector based signal bus, where the bus vector merges together all the terminal vectors. Although, this bring in a problem with always keeping in mind the correct number of indexes dependent on the number of connected components.

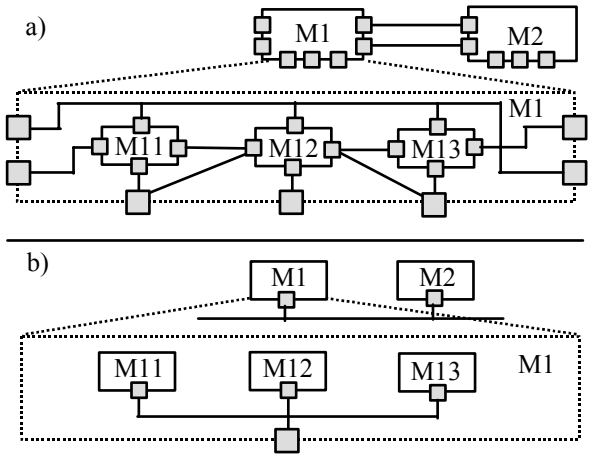

Figure 1. A combination of interconnections with different types of terminals at several levels (a) and a general signal bus using one type of terminal enabling a plain structure (b)

Even the index of each component has to be determined if a general approach is used. In this case it is necessary to define the exact number of signals in every terminal to be connected and the bus needs a pre-defined vector index according to number of connected components. If a uniform terminal is used with a large number of signals where several models are not using but a small amount of terminal signals, the SE will have an unnecessary high number of signals to handle. It is therefore desirable to be able to choose which internal module signals to be exposed by limiting the terminal signals. The goal of a general structure is a signal bus that automatically assign the components with their identities and that enables the user to mainly focus on the signal to be exchanged and not on the under laying

structure and functions that manage the signal communication. The structure should be adaptive to different user specific desires and also able to resemble several types of communication, e.g. between software processes, computers or in automation systems. One solution based on Modelica interconnected to a C++ library will be further described and exemplified.

2. Design

The design is meant to work both in a SE and for real applications. This calls for either an automatic translation from the modeling language to the target application or a smooth software interface between the SE and the chosen program language. As the Modelica language has nice facilities for external function calls the main functions of the library is implemented in a C++ library [3,4,5].

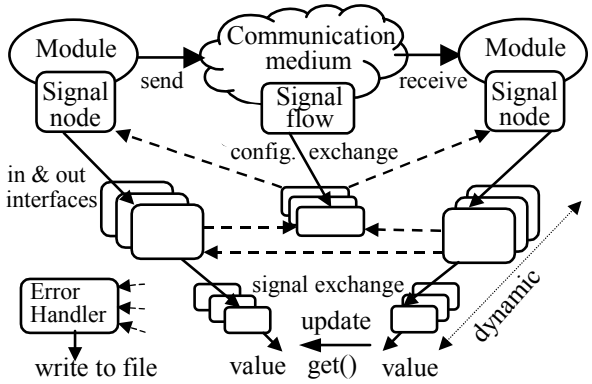

Figure 2. Signal management design overview

The basic idea is that independent of different types software modules there should be simple means to establish signal configuration and connection via some type of communication media to another module as depicted in Figure 2. A module could be a software process, thread or a model in a SE where the communication media could be an external or internal communication link [6]. The design is divided in a hardware and software structure, where the physical part, communication lines, transmitters and receivers are developed in the Modelica languages and the software functions in C++. The principal structure in Modelica builds upon signal nodes and signal flows where the signal nodes may represent a temporary storage, transmitter or receiver. The signal flows represent the communication lines between the signal nodes with the main tasks, in initiating mode, to inform the connected nodes about the configuration and, in operation mode, to control that the physical line is in order for signal transmitting. The signal nodes creates in- and output interfaces according to the signal flow configurations as shown. The interfaces in turn create individual signal objects for each specified signal. An input interface then points to an output interface of another signal node where the configurations are checked before switching to operation mode. During operation, each signal object updates the data when trigged by the signal node. Every failure or configuration error is reported to an error manager that writes the needed information in a file or to the log window in the SE.

3. Signal classification and configuration

An important issue concerning signal classification is how to enable several types of configurations without making it too complex. The following signal classifications are used to configure the signal flows.

- The ***Signal Identity*** (SI, SIdentity), which enables a signal to have a unique identity, but this might imply obstacles regarding dynamical capabilities
- The ***Signal Type*** (ST, SType), where each signal must be specified by a unique type (e.g. command, power set point, etc)
- The ***Signal Block Type*** (SBT, SBType): a predefined number of signal types, which could be uniformed (protocol)
- The ***Signal block Group Identity*** (SGI, SGIdentity) can be used if several SBTs are connected between the same source and destination. The unique group identity enclosures SIs, STs or/and SBTs.

Signal flow configurations must at least have a specification on a SIdentity or SType. Normally the SIdentity is used as a unique identity that can be found anywhere in a system model. However, if using a model with several components of the same type and signal interface together with a higher-level control unit collecting and distributing signals, the SBTypes and SGIdentities are requested as depicted in Figure 3.

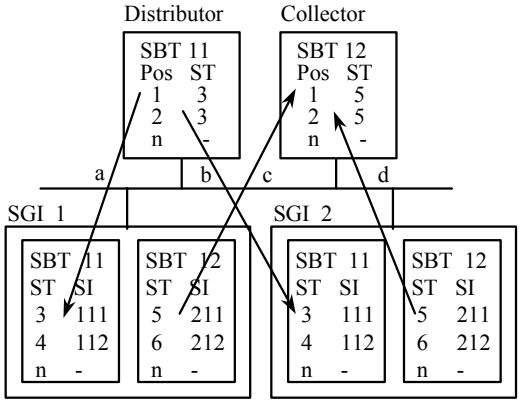

Figure 3. Example of the signal classification and their signification

First, by examine each component it is desirable to use the same set of identities for the signals in each component. As in this case there is at least two SBTypes (11 & 12), one for input and one for output signals, where each component signal flow need to be unique by enclosure all the SBTypes in a SGIdentity as shown. Further, it is possible to use only the STypes and not the SIdentities at higher level, as the intention is a dynamic

distributor and collector at the higher level. Arrow "a" and "b" in the Figure point out that it is the SType 3 at position 1 and 2 that should be transmitted to SGIdentities 1 and 2 respectively. For the collector it is similar the SType 5 at position 1 for both SGIdentities 1 and 2 that should be transmitted to the two first positions of the collector. If a new component is added with SGIdentities 3, the distributor and collector per automatic should extend the vector signal with respective STypes.

4. Modelica library

The signal classifications are the basic parameters for configuring the signal flows in the SE. The SignalFlow library is built in Dymola, using the Modelica language. Dymola is an object-oriented SE for modeling transient physical systems that has god support for interconnecting other object-oriented languages such as C++, which is well exploit in this library [3,4,5,7].

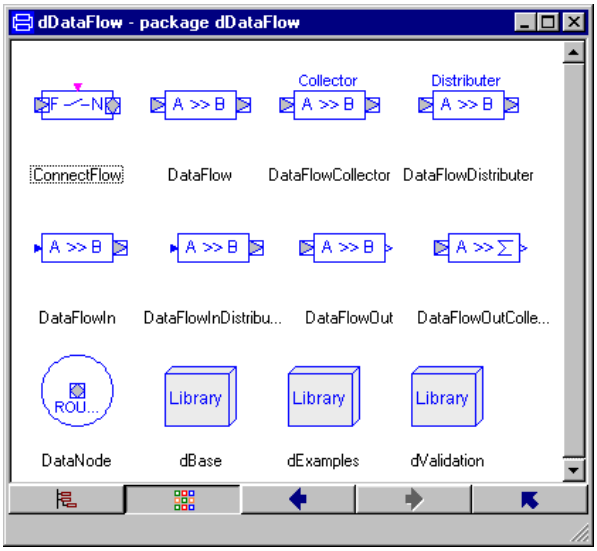

Figure 4. The SignalFlow library

The SignalFlow library is depicted in Figure 4 and mainly contains the following component models:

- The ***SignalNode*** (SN, SNode) model manage all in and out coming signal interfaces initiated by the SignalFlows
- The ***SignalFlow*** (SF, SFlow) model represents the signal configuration between the SNodes.
- The ***SignalFlowOut*** (SFO, SFlowOut) model has a standard Modelica output terminal to interconnect with other library models.
- The ***SignalFlowIn*** (SFI, SFlowIn) model has a standard Modelica input terminal that supplies the signal system with signals.
- The ***SignalFlowDistributer*** (SFD) model is similar to the SFlowIn but initiates an automatic search for SFlowOut models that are configured according to a predefined SType.
- The ***SignalFlowCollector*** (SFC) model initiates an automatic search among SFlowOut models for signal types to be collected (model with sum-sign).
- The ***SignalResourceManager*** (SRM) model interacts by connecting a SNode where it accesses a predefined resource type.

The main component models are constructed several sub libraries within the Base library, which contain the dConnector, dInterface, dIcon, dRecord, and dFunction library. The sublibrary dConnector contains two connector (terminal) types that are defined by the following Modelica semantics:

```
connector SignalNodePort
    Real   signalNode;
    flow   signalLine;
end  SignalNodePort;
```

The second connector "SignalFlowPort" is identical except for the icon, which is a triangle instead of a quadrangle as in the "SignalNodeConnector" case. The model interfaces, within the dInterface sublibrary, are composed as below where the node and flow have one two connectors respectively.

```
partial model SignalNodeInterface
  extends dIcons.SignalNodeIcon;
  dConnectors.SignalNodePort nodePort;
  Real nodeS  = nodePort.signalNode;
  Real lineS    = nodePort.signalLine;
end SignalNodeInterface;

partial model SignalFlowInterface
  extends dIcons.SignalFlowIcon;
  dConnectors.SignalFlowPort flowPortA;
  dConnectors.SignalFlowPort flowPortB;
  Real nodeA  = flowPortA.signalNode;
  Real lineA    = flowPortA.signalLine;
  Real nodeB = flowPortB.signalNode;
  Real lineB    = flowPortB.signalLine;
end SignalFlowInterface;
```

The user interface assigning the SNode parameters are depicted in Figure 5 where the node type can be used to force the node to be of storage type. The node identity is normally automatically assigned but could also be forced to a specific value, and the last parameter determines if the node should be continuously or discrete. The SNode in the SignalFlow library is initiated and automatic assigned an identity by the zNodeInit function. The function argument is a configuration vector (cv) where all predefined parameters are placed. Dependent on whether the SNode should be discrete (sampled) or continuous the variable nodeD or nodeC is assigned. When initiated the node updates every time interval according to the SE and extended equations are only one technique solving the discrete or continuous options. The SNodes are always initiated at simulation start and during simulation the only input is the "lineS" that is one of the arguments in the zNodeUpdate function.

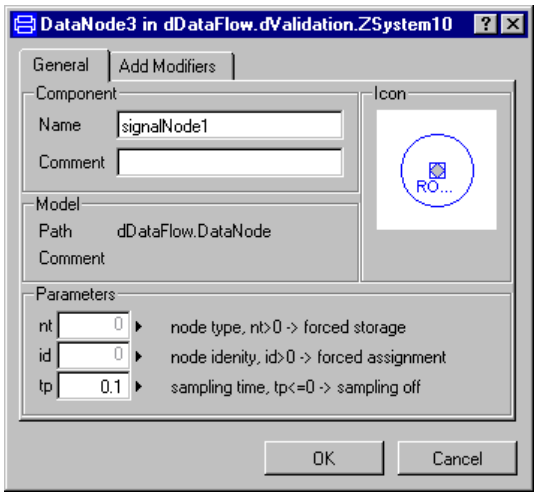

Figure 5. User interface for assigning the parameters of the SignalNode model

As the "lineS", per Modelica-definition, is declared as a "flow", all connected SFlow models are summarized in this variable enabling the node to examine which lines that are active.

```
model SignalNode
   extends dRecords.SignalNodeRecord;
   extends dInterfaces.SignalNodeInterface;
protected
   Real nodeID(start=0.0);
   Real nodeC(start=0.0);     // continuous
   Real nodeD(start=0.0);     // discrete
   Boolean sampleTrigger;
equation
   when initial() then
     nodeID = dFunctions.zNodeInit(cv, cvSize);
     reinit(nodeC, nodeID);
   end when;
   sampleTrigger = if samplingON then
         booleanPulse1.outPort.signal[1] else false;
   when sampleTrigger then
     nodeD = dFunctions.zNodeUpdate(nodeID,
               lineS,  time);
   end when;

   der(nodeC) = if samplingON then 0.0 else
                  nodeID – dFunctions.zNode_
                  Update(nodeID, lineS, time);
   nodeS = if samplingON then pre(nodeD)
           else nodeC;
end SignalNode;
```

By definition, the SFlow model has always a flow of signals from A to B as shown by the icon and in the "SignalFlowInterface" declaration where the two terminals are denoted A and B. The SFlow model is initiated as soon as the variables "nodeA" and "nodeB" are positive. The initiating function, zFlowInit, then automatically returns the line identities lineAID and lineBID. In normal operation the SFlow only checks that the line is correct for transmitting. If a failure occurs on the line, the ""lineA" and "lineB" are assign to an error code. When the line is restored the initiating process once again is performed.

```
model SignalFlow
   extends dRecords.SignalFlowRecord;
   extends dInterfaces.SignalFlowInterface;
   Real flowID(start=0);
   Real lineAID(start=0);
   Real lineBID(start=0);
equation
    when (nodeA*nodeB > 0) then
      flowID = dFunctions.zFlowInit(nodeA, nodeB,
                cv, cvSize, signalType, sTypeSize,
                signalAID, sAIDSize, signalBID,
                sBIDSize);
      lineAID = dFunctions.zTryConnectFlowOut(flowID,
                 nodeA, time);
      lineBID = dFunctions.zTryConnectFlowIn(flowID,
                 nodeB, lineAID, time );
 end when;
 lineA = dFunctions.zFlowUpdate(flowID, nodeA,
          lineAID, nodeB, lineBID, time);
 lineB = dFunctions.zFlowUpdate(flowID, nodeB,
          lineBID, nodeA, lineAID, time);
end SignalFlow;
```

In the SignalFlowOut model the lineB is not used and in the SignalFlowIn model the lineA is not used. They are replaced by the "value" variable that is connected to the standard Modelica Input or Output connectors.

```
 ----------------------------------------------SignalFlowOut ---
 lineA = zFlowUpdate(flowID, nodeA, lineAID);
 for index in 1:sTypeSize loop
   value[index] = zFlowGet(nodeA, lineA, index);
 end for;
 ----------------------------------------------SignalFlowIn ----
 lineB2 = zFlowUpdate(flowID, nodeB, lineBID,…);
 lineB  = zFlowSet(nodeB, lineB2, value, valueSize);
```

In the zFlowInit function, the argument is equivalent to the "SignalFlowRecord" that corresponds to the signal classification in section 3 and in Figure 6.

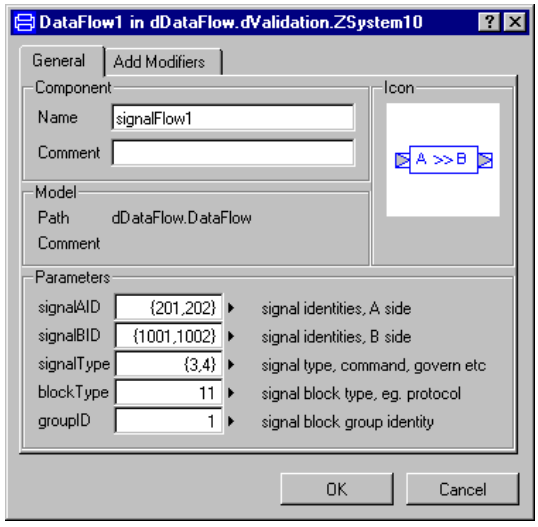

Figure 6. User interface for assigning the parameters of the SignalFlow model

Both side A and B are represented with one identity vector each enabling different identity assignments of the same signal in different communication areas. Using the library in the simplest case the identity is the only parameter to be assigned.

```
record SignalFlowRecord
   parameter Integer[:] signalAID ={-1};
   parameter Integer[:] signalBID ={-1};
   parameter Integer[:] signalType={-1};
   parameter Integer groupID;
   parameter Integer blockType;
protected
   parameter Integer[:] cv={0,0,0,groupID,blockType};
   parameter Integer cvSize=size(cv, 1);
   parameter Integer sTypeSize=size(signalType, 1);
   parameter Integer sAIDSize =size(signalAID, 1);
   parameter Integer sBIDSize =size(signalBID, 1);
end SignalFlowRecord;
```

The SNode model has no direct limitation in connecting the number of SFlow models, as the structure is dynamic. Each new connection creates a new object that might be automatically removed if disconnected a predefined amount of time. The structure is simple and can be connected with unlimited SFlows between SNodes, Figure 7, and at any hierarchy level. In the simplest case one SNode is used as a communication bus where all SFlows are connected to that single bus. The SFlows are normally embedded in some user specific model hiding the pre-defined communication interface interacting the bus.

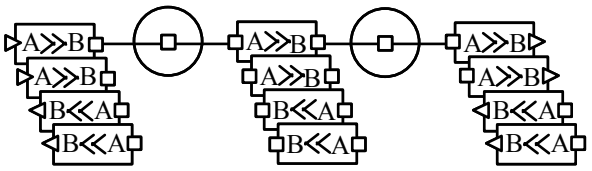

Figure 7. Dynamic number of connections

The basic Modelica structure for connecting SNodes and SFlows, depicted in Figure 8, prevents algebraic loops, which is easily caused with a high degree of control levels in a SE. Even if the SE can handle this, as in the Dymola case, it might become a complicated problem in large system models. The SNodes have a state (node) corresponding to capacitors (voltage) summarizing the variables from the SFlow models. The SFlow assigns the lineA and lineB, identities, corresponding to currents in the electrical case. The line identities are then identified by using a "modulo 2" function both when assigning and decoding the identities, e.g. if four SFlows are connected and the identities of them are 2, 4, 8 and 16 with the sum of 30, it is easy for the SNode to decode the SFlows to determine both if a new SFlow has to be configured and if a line is broken. Referring to Figure 8, each node can be connected to numerous signal flows where the SNode assigns the terminal variable "nodeS" and the line identities are summarized in the terminal variable "lineS" of the node. In the SFlow, the lineA and lineB are separated enabling the responsible node to change the line identity if needed. In case of disconnecting one SFlow model, it is not likely that the identities will be the same in a dynamic environment. There is also a cross coupling between the nodes by the arguments in the update function, which uses the corresponding node identities in order to avoid losing the equation tail in the SE. The SE initiating process automatically assigns all identities of the SNode and SFlow models.

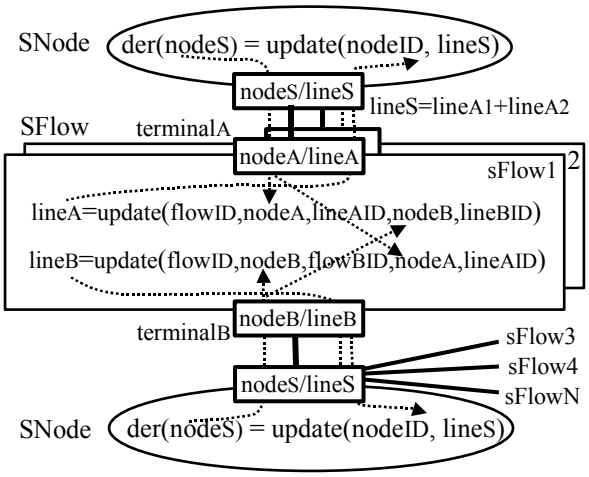

Figure 8. Terminal assignments between SNodes and SFlows

5. Interconnection to C++ library

The Modelica models in the SignalFlow library have several functions for interacting the with C++ library. All included functions are similarity declared as here exemplified for the "zNodeUpdate" function.

```
function zNodeUpdate
  annotation (Library={"Libcore"});
  input Real nodeID;
  input Real flowSum;
  output Real y
external "C" y = zNodeUpdate(nodeID, flowSum);
end zNodeUpdate;
```

The "Libcore" assignment is the actual C++ library that is linked to the SE by the "Libcore.lib" file, which is placed under the "dymola\bin\lib" directory.

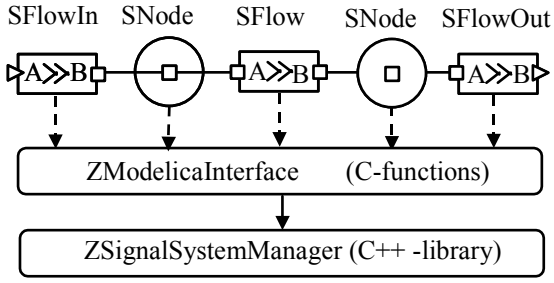

Figure 9. Modelica and C++ interconnection

Each SNode and SFlow model creates a separate object in the C++ layer where the Modelica-layer models use function calls to the ModelicaInterface and method calls to the C++ library, as depicted in Figure 9. The header declarations for the "ZModelicaInterface.h", enables method calls from the C++ implementation, as shown below. By using the "ZModelicaInterface.cpp", the Modelica function calls are translated to method calls from the "ZSignalSystemManager" class where all methods can be found in each step interfacing the C++ library.

```
// --- ZModelicaInterface.h ---
#ifndef ZMODELICAINTERFACE_H
#define ZMODELICAINTERFACE_H

#include "ModelicaUtilities.h"
#include "ZSignalSystemManager.h"

#ifdef __cplusplus
extern "C" {
#endif
double zNodeInit(int cv[], int cvSize);
double zNodeUpdate(double nodeID, ….);
……. remaining functions

#ifdef __cplusplus
}
#endif
#endif  // end
```

The description of the specification is here exemplified, where the remaining functions are declared as the two presented.

```
// --- ZModelicaInterface.cpp ---
#include "ZModelicaInterface.h"

ZSignalSystemManager    sys;  // C++ class

double zNodeInit(int cv[], int cvSize)       {
   return sys.nodeInit(cv, cvSize);
);
double zNodeUpdate(double nodeID,...)  {
   return sys.nodeUpdate(nodeID, ….);
}
……. remaining functions
// end
```

The "ZSignalSystemManager" class is the actual interface between the Modelica-layer that manages all the ZSignalNode and ZSignalFlow objects using dynamic vectors as is briefly shown here. The JVector class is a dynamic vector (DV) equivalent to the CVector class in the C++ standard library except for some modifications making a smooth conversion to the JAVA environment. The dynamic properties in the library are based on the DV that is used for pointing out all needed objects for the specific application. There are facilities in Modelica allowing to declare void*-pointers and external objects by defining a partial class "ExternalObject" with constructor and destructor functions that would make the ZSignalSystemManager excessive. However, this is not used in this version but might be implemented in the next.

```
typedef JVector<ZSignalNode*> ZSignalNodeVector;
typedef JVector<ZSignalFlow*>  ZSignalFlowVector;

class ZSignalSystemManager
{
private:
   ZSignalNodeVector*  m_nodeVector;
   ZSignalFlowVector*   m_flowVector;
   int   m_nodeCounter, m_flowCounter;
   ZOutFileManager*     outfile;
public :
   ZSignalSystemManager();
   ~ZSignalSystemManager();
   double nodeInit(int cv[], int cvSize);
   double nodeUpdate(double node,  flowS,  time);
   double flowInit(double nodeA, nodeB, int ……...);
   double flowUpdate(double flow, nodeA, ...……..…);
   double tryConnectFlowIn(double flow, node, flowA);
   double tryConnectFlowOut(double flow, node);
   double flowSet(double node, flow, *value, ……..…);
   double flowGet(double node, flow, int index);
   void outputManager(const char* text,int type);
}; // end ZSignalSystemManager
```

6. The C++ class library

The key to manage the dynamic design interconnected to a SE is to use higher levels of abstraction as in object-oriented languages [6,8]. An example for this approach is implemented in a C++ class library based on implementation corresponding to the SNode and SFlow models used in the Modelica layer.

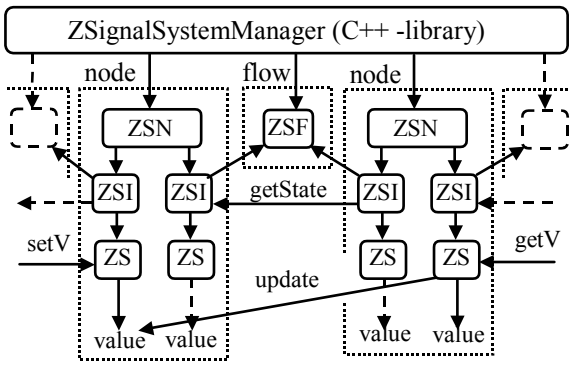

Figure 10. C++ structure

Using the C++ class libraries, a designer can describe components at a broad range of abstraction levels, which result from the ability to perform signal and control modification separately. As pointed out in section 1, there are different levels of abstraction at which C++ can be used for the signal management system as depicted in Figure 10. The C++ library mainly includes the following classes:

- The ***ZSignalNode*** (ZSN, ZSNode) class is responsible for searching and joining together every signal in each connected input and output signal

interface. The class is receptive to changes of new interconnections and continuously controls all connections for not being defected.

- The ***ZSignalInterface*** (ZSI, ZSInterface) manages a pre-defined number of signals in a block that, for example, could correspond to a protocol. This object could be of several types such as inputs and outputs for communication but also parameters for configurations and specifications.
- The ***ZSignal*** (ZS, ZSignal) class is the object containing the particular value of the signal and its configuration. It could also be a reference pointing at another ZSignal.
- The ***ZSignalFlow*** (ZSF, ZSFlow) class is the actual configuration of the signals between two SNodes including the needed types and identities.

As depicted in Figure 10, the SNode model is interconnected to the ZSNode class and the SFlow model to the ZSFlow class. The ZSNode includes DVs pointing at ZSInterface objects that in turn also have DVs pointing at ZSignal objects.

```
class ZSignalNode
{
private:
   ZIntegerVector*            m_cv;
   ZSignalInterfaceVector*    m_iv;
   ZSignalInterfaceVector*    m_ov;
   int                        m_state;
   bool                       m_change; m_storage;
public:
   static int m_signalNodeCounter; // object counter
   // constructor and destructor
   ZSignalNode(int cv[], int cv_size);
   ~ZSignalNode(void);
   // --------- configuration functions ------------------------
   int     verifyInInterface(int cv[], int cv_size);
   int     addInInterface(ZSignalFlow* flow, doub. time);
   void    removeInInterface(int iID);
   //  corresponding functions for OutInterface
   void    tryConfiguration(void);
   void    configureAllSignals(void
   void    verifyAllSignals(void);
   void    findSignalByID(ZSignalInterface* di);
   void    findSignalByType(ZSignalInterface* di);
   void    findSignalByBlock(ZSignalInterface* di);
   // ----------- operational functions ----------------------
   void     update(int nodeID, double ntime);
   void     setValues(interfaceID, dou* value, int size);
   double getValue(int interfaceID, int index);
   // ----------- error and information functions ----------
   void    checkConfiguration(int cv[], int cv_size);
}; // end ZSignalNode
```

At simulation start, the ZSNode is created and initiated. A unique identity is then returned to the SNode model, which represents a pointer to the ZSNode object. The ZSNode object is at start in configuration state and awaits method calls for SFlow connections. As soon as the SFlow model is properly interconnected, it attempts to call respective ZSNodes by the "zTryConnectFlow" method. This checks the SFlow reference and then calls for the "addInInterface" or the "addOutInterface" of the ZSNode object dependent on if connected to the A- or B-side. The "addInterface" method then creates a new ZSInterface object according to the configuration of the ZSFlow model and returns a unique line identity according to the ZSInterface object. Every time a new event occur in the SNode, the internal variable "change" is set, which start internal methods to find in- and output signals that are matching and then connects them dependent on configurations. As soon as the ZSInterface is correct interconnected, it is turned over to "operation" state and starts updating the signals continuously or according to a sample rate. Dependent on output interface configurations the ZSNode has methods to find input signals according to signal identity (findSignalByID), or type (findSignalByType). In the type case, there are also more specific methods searching particular block types. This is, for example, used when the output interface is of collection type, which is expanding according to the number of input interfaces including the requested signal type. In operation state the ZSNode only uses the "update" method that propagates the call to the affected ZSInterface objects.

```
class ZSignalInterface
{
private:
  // Internal variables
  ZIntegerVector*   m_cv;
  ZSignalFlow*      m_flow; // configuration
  ZSignalVector*    m_sv;
  ZSignalInterface* m_siConnected;
  Int               m_state
  bool              m_active, m_change;
  bool              m_storage, m_destination;
public:
  static int m_signalInterfaceCounter;  //  object counter
  //  constructor and destructor
  ZSignalInterface(int cv[], int cv_size, ZSFlow* flow);
  virtual ~ZSignalInterface();
  // ---------- configuration functions ---------------------------
  int          getSize(void);     // number of signals
  int          getType( void );  // interface type
  int          getStorageType(void), getState(void);
  void         setState(int state);
  int          getGroupID(void);
  ZSignal*  getSignalRefByIndex( int index );
  ZSignal*  getSignalRefByID( int id );
  ZSignal*  getSignalRefByType( int type );
  void         connect(ZSignalInterface* di);
  void         tryConfiguration(void);
  void         verifyConfiguration(void);
  void         addSignalElement(ZSignal* sObject);
  void         removeAllSignalElements(void);
  bool         change( void );
  // -------  operational functions ------------------------------
  void         update(void);
  void         setValues(double* value, int valueSize);
  double     getValue(int index);
  double     getSum(void), getMax( void ), getMin( void );
  // ------- check functions ( throwable) ----------------------
  void         checkConfiguration(int cv[], int cv_size);
  }; // end ZSignalInterface
```

The ZSInterface is based on a vector (m_sv) including a pre-defined number of ZSignal objects. The ZSInterface has several functions for configuration, mainly to interconnect and verify all included ZSignal objects to the intended destinations for updating the flow when turning to operation mode. If the ZSInterface object is of input type it has also a reference (m_siConnected) to an output ZSInterface of another ZSNode object. This gives that an input ZSInterface can never leave the configuration state until the reference-pointer points at an output ZSInterface in operational state. This is the actual control chain that implies that the source ZSInterface is the first one turning in operational state and then, step by step, permits the chain of ZSInterfaces to the last instance, the destination ZSInterfaces, to become operational. In the ZSInterface class, there are additional methods (getSum, getMax, getMin) normally used if utilizing the collector type, SFC model, which collects specific signal types.

The bottom class, ZSignal, includes the value of one signal and its signal specification. There are built-in functions to determine whether the signal should be locally stored or only point at another ZSignal object. The ZSignal class and its methods are shown beneath.

```
class ZSignal
{
private:
  int          m_signalID,  m_signalType;
  int          m_blockType, m_groupID;
  double       m_value;
  double*      m_valueRef;
  bool         m_refOK, m_storage;
public:
  ZSignal( int bid, int id, int type, bool storage );
  virtual ~ZSignal();
  // --------- configuration functions -----------------
  void         setValueRef(double* value);
  double*      getValueRef(void);
  bool         refOK(void);
  bool         active(void);
  int          getGroupID(void);
  int          getBlockType(void);
  int          getSignalType(void);
  int          getSignalID(void);
  // ----------operational functions --------------------
  void         update(void);
  void         setValue(double value);
  double       getValue(void);
  // --------- check functions (throwable) -----------
  void         checkConfiguration();
}; // end ZSignal
```

The update, setValue and getValue methods are the actual methods that the higher-level classes call for in operation state. Consequently, all signal management is only handled between the ZSignal objects that, in fact, are not aware of the other classes. They are only utilized to keep track and be prepared to change connections according to the signal configurations and routings.

7. Verification by samples

The "SignalFlow" library is verified by using all the components in several connections as depicted in Figure 11, which corresponds to the configuration example in Figure 3. At the left hand, there are two identical areas with internal control using the signal facilities. The SGIdentities are assigned 1 respective 2 that in this case also represents the two units. Unit 1 is not connected until 0.3 second after simulation start for testing of components added during simulation. Each unit has a SNode (SN) corresponding either to a communication intermediate storage area or a complete database for the unit where a number of signals are selected. All SNodes are assigned to 10 Hz sample rate. The SFlowOut of unit 1 and 2 are only assigned with the SIdentities and the SFlow models are assigned with SGIdentities, SBTypes, STypes and SIdentities enabling SFDistributors and SFCollectors to be used. The SFDistributor distributes two signals that, at start, are assigned 1.0 and 2.0. Reaching 0.8 second, the signals are increased 0.5 to 1.5 and 2.5.

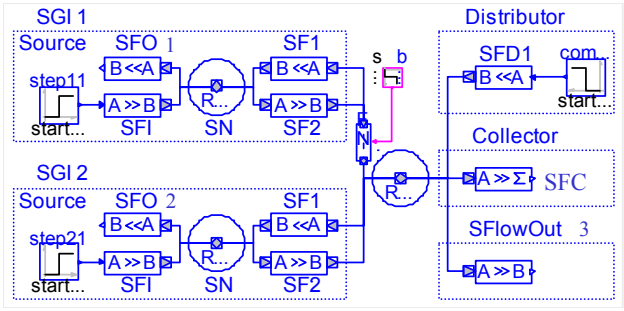

Figure 11. Signal system setup for validation of several different signal exchange possibilities

The signals are distributed to the SFlowOut (SFO1, SFO2) models of respective unit, which is depicted in the upper graph in Figure 12, where the initiating for configuration requires 3 samples. At this point, the SFlowOut2 is in operation state and updates the signal to the value 1.0. The SFlowOut1 should have been operated in the same manner but is not connected until time equal to 0.3 second, which then takes another 2 samples to be configured before turning to operating state. At 0.8 second, the two signals are directly increased to verify that there is no delay time in operation state. This example shows that the SNodes can handle altered configuration during simulation with only a few samples of delay and that the signals are distributed to the intended units. Moreover, in the other signal flow direction, the sources of unit 1 and 2 are constant 1.0 respectively 2.0. The SFlowOut3 are configured by SIdentities to connect these to signals. In the middle graph, Figure 12, this is also shown by first being delayed 3 samples before operating the signal from unit 2 and then additionally 2 samples for unit 1 due to the afterward connection at 0.3 second. Assigning the SBTypes and STypes configures the SFCollector in the third case. The bottom graph in

Figure 12 shows that the configuration events are the same as in the previous case. The value 1-3 corresponds to the sum, max and min functions, which are correct at 0.5 second, where the sum is equal 1+2=3, max=2 and min =1. However, this example is configured with a small number of SNode and SFlow models that imply that few samples are needed for the configuration state. Consequently, in more complex system models the configuration sample delay increases but not necessary in time, dependent on the sampling rate.

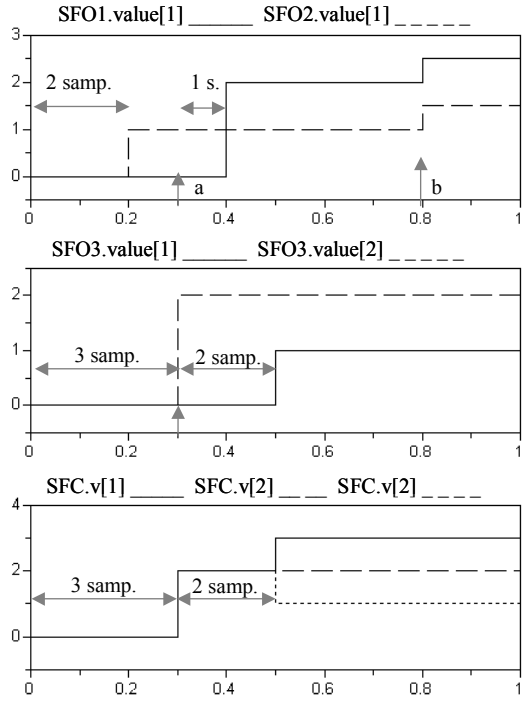

Figure 12. Simulation results for the validation model

A more complex illustration, where the SignalFlow library is frequently used is depicted in Figure 13, which represents a modular wind power plant (WPP) model in Dymola. The model is built from several model libraries, the "SignalFlow", the "ControlFlow" mainly including control system related units, the "PowerFlow" (converters, cables etc.) and the "WindPower" library (wind turbine units). The WPP model has several control unit levels and consequently several communication levels (CL). The four CLs included are the production control level (CL4), plant control level (CL3), process control level (CL2) and field control level (CL1). The principles are similar at each level including several controllable power units connected to a control unit via a communication bus. At each level, the SignalFlow library is used for exchanging signals. A single SNode represents different types of communication buses (B) dependent on the CL, and is also used for inter-process communication within a control unit. Between each CL, a control unit is interconnected that contains two signal exchange (E) units (SNode, SFlows) separating the CLs, and a number of software modules (M) controlling the connected units. A module includes several function blocks (FB) that are connected by the SFlowIn and SFlowOut models to a SNode representing the local database. A selected number of signals are then exchanged from the database to the external bus using two SFlow models.

8. Conclusion

An adaptive signal management structure, based on object-oriented dynamical programming, is presented. A simulation library, "SignalFlow", is developed in Modelica, where the model components are interconnected to the signal management structure. The presented results verifies that the structure meet the requirements. The structure forms a base level layer enabling adaptation for higher-level control and information flows. It is also a good example on how to develop function calls, interacting external programming languages with a simulation environment.

References

[1] Svensson, J. and Karlsson, P., "Wind Farm Control Software Structure", *Third International Workshop on Transmission Networks for Offshore Wind Farms*, Royal Institute of Technology, Stockholm, Sweden, April 2002

[2] Svensson, J., Karlsson, P. and Johnsson, A., "Information Structures for Scalable Distributed Power Systems", *3:rd IASTED International Conference on Energy and Power System*, Marbella, Spain, September 2003

[3] Freiseisen W.; Keber R.; Medetz W.; Pau P., Stelzmueller D.,"Using Modelica for Testing Embedded Systems", *2:nd International Modelica Conference*, Proceedings, pp. 195-201

[4] Pereira Remelhe, M.A.,"Combining Discrete Event Models and Modelica - General Thoughts and a Special modeling Environment", *2:nd International Modelica Conference*, Proceedings, pp. 203-207

[5] Modelica™ – *A Unified Object-Oriented Language for Physical Systems Modeling, Languages Specification, Version 2.0*, http://www.modelica.org

[6] Svarstad, K.; Ben-Fredj, N.; Nicoleson, G.; Jerraya, A., "A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems", *Conference on Asia South Pacific Design Automation 2001*, Yokohama, Japan

[7] Dymola, Dynamic Modeling Laboratory, Dynasim AB, Lund, Sweden, http://www.dynasim.com

[8] Al-Agtash, S., Al-Fayoumi, N. "*A Software Architecture For Modeling Competitive Power system"*, IEEE Transactions on Power Systems, 2000, pp. 1674-1679

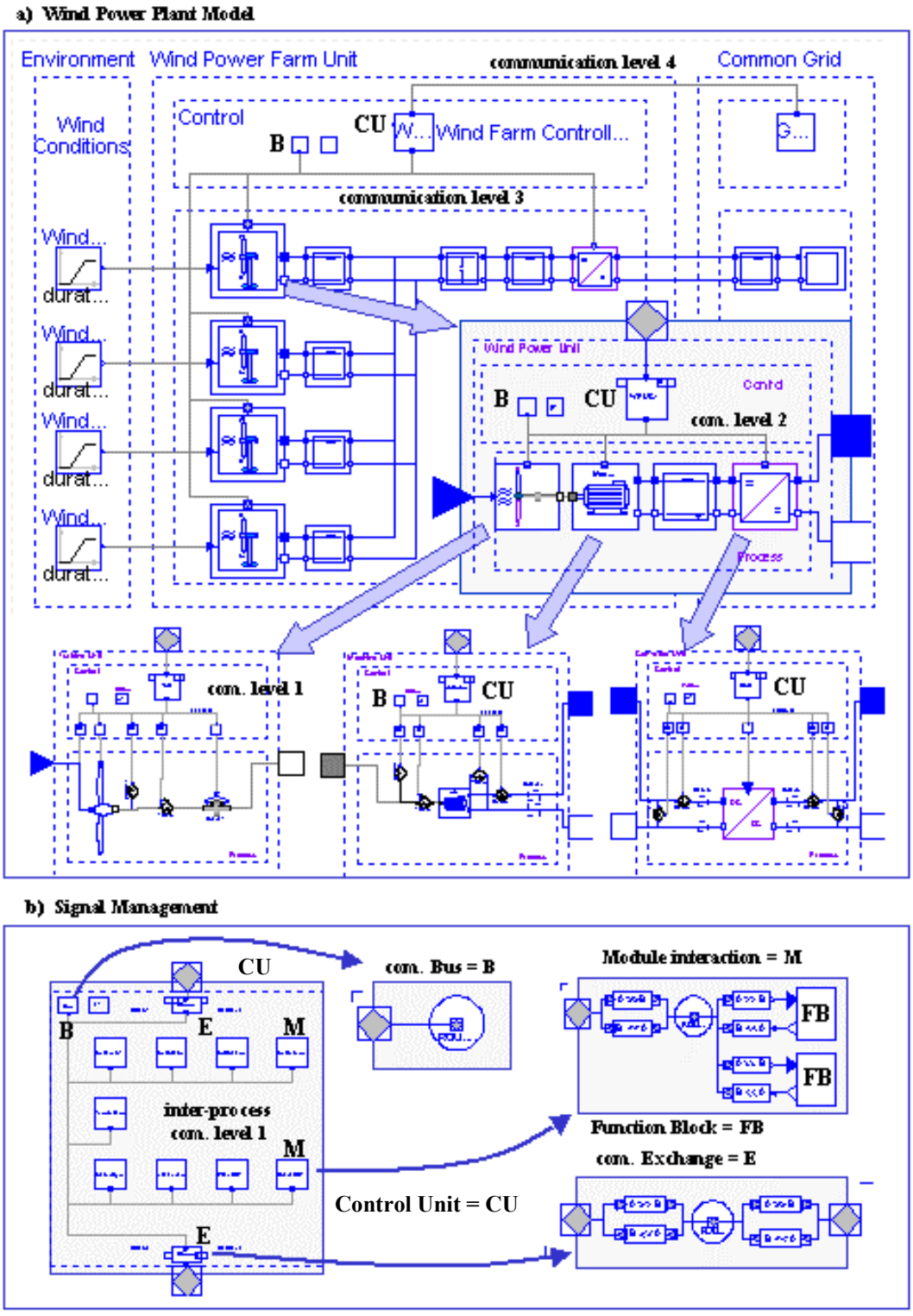

Figure 13. Modular simulation model of a Wind Power Plant including several communication levels (a), and signal management models used in the WPP model (b)