Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

L. Saldamli, B. Bachmann, P. Fritzson, H. Wiesmann
Linköping University, Sweden; FH Bielefeld, Germany; ABB, Switzerland

A Framework for Describing and Solving PDE Models in Modelica

pp. 113-122

A Framework for Describing and Solving PDE Models in Modelica

Levon Saldamli*, Bernhard Bachmann†, Hansjürg Wiesmann‡, and Peter Fritzson*

Abstract

Currently, the Modelica language [3, 4] has limited support for solving partial differential equations (PDEs). There is ongoing work for introducing PDE support at the language level [5, 6]. This paper describes a prototype for describing PDE problems using the Modelica Language without any extensions, as an intermediate step. The goal is to define standard PDE models independent of specific domains, boundary conditions or any spatial discretization, and allow a user to reuse this without manual discretization. Modelica packages are used to define continuous domain boundaries, domains, and field variables over domains. Corresponding space discrete version of these packages are used to solve the space discretized PDE problem.

1 Introduction

A PDE problem can be specified and solved as follows using the approach in this paper (see Section 6 for more details):

```
model GenericBoundaryPoissonExample
  parameter BoundaryCondition.Data dirzero(
    bcType=BoundaryCondition.dirichlet,
    g=0);

  parameter BoundaryCondition.Data dirfive(
    bcType=BoundaryCondition.dirichlet,
    g=5);

  package myBoundaryP = MyGenericBoundary;
  parameter myBoundaryP.Data mybnd(
    bottom(bc=dirzero),
    right(bc=dirfive),
    top(bc=dirzero),
    left(bc=dirfive));

  package omegaP = Domain(
    redeclare package boundaryP=myBoundaryP);
  parameter omegaP.Data omega(boundary=mybnd);
  package PDE =
    PDEbhjl.FEMForms.Equations.Poisson2D(
      redeclare package domainP = omegaP);
  PDE.Equation pde(
    domain=omega,
    g_rhs=1);
end GenericBoundaryPoissonExample;
```

*Dept. of Computer and Information Science, Linköpings universitet, Linköping, Sweden. *{levsa,petfr}@ida.liu.se*

†Fachhochschule Bielefeld, Fachbereich Mathematik und Technik, Studiengang Mathematik, Bielefeld, Germany. *bernhard.bachmann@fh-bielefeld.de*

‡ABB Corporate Research, CH-5405 Baden, Switzerland. *hj.wiesmann@bluewin.ch*

First, two Dirichlet boundary conditions are declared, `dirzero` and `dirfive` with the right-hand side values 0 and 5, respectively. Then, a boundary component `mybnd` of type `MyGenericBoundary` (see Section 6) is declared, and the boundary conditions are assigned to the boundary components `bottom`, `right`, `top` and `left` of `mybnd`. A domain named `omega` is then declared using the boundary object `mybnd`. Finally, the PDE model is instantiated using `omega` as its definition domain.
Each declaration requires two actual declarations, one for the package and one for the data of the object, as explained in the following section.

2 The Package Approach

In order to use an object-oriented approach with polymorphism in Modelica, we use packages for defining new types such as `Domain`, `Field`, and `Boundary`. Each type `Type` contains at least a record called `Data`, containing the member variables needed in objects of type `Type`. The member functions are declared in the `Type` package. Each member function has at least one input argument, of the record type `Type.Data`. Thus, when calling member functions on objects, the objects data is passed as the input record argument. Declaring an object of type `Type` is implemented by declaring a local package, e.g. `typeP` which extends `Type` and possibly modifies parts of it, and then declaring a component of type `typeP.Data` which contains the data of the object of the modified type. This way, replaceable functions can be declared in a package, and replaceable packages extending these can exchange the functions as required. In other words, the packages define the class hierarchy for lookup of functions to work, and the data records define the object hierarchy storing the object instance data.

When types contain instances of other types, they declare a local package extending the other types package, and declare the objects data inside the `Data` record. For example, an equation model declares a domain and a field as follows:

```
model Equation "Poisson equation 2D"
  replaceable package domainP = Domain;
  parameter domainP.Data domain;

  package fieldP = Field (
    redeclare package domainP = domainP);
  parameter fieldP.Data u(domain=domain);
end Equation
```

This approach allows the equation model to be reused with any domain without changing other parts of the model. The package `fieldP` redeclares the replaceable domain package in the `Field` package. This way, the package hierarchy is correctly set up. The actual data records are declared separately, in order to build the object hierarchy of the model. The record *u* has the type `fieldP.Data` and will contain the correct domain data type from the given package `domainP`. The domain data must be initialized with the local values though, which is done with the modification when declaring the record `u`. When the domain is to be discretized, the shape function of its boundary package is called. Since the boundary package of the domain package is replaced when the domain is declared, the correct shape function is called. This is handled automatically through the package `DiscreteDomain`, explained in Section 5.1.1, which is declared in the discrete parts of the equation models.

The drawback of this approach is that each instantiation requires definition of a local package extending the type package, together with a declaration of the `Data` record of the local package. The advantage is that the local package can be declared as replaceable, and the correct version of the package will be used without knowing the type in advance.

3 Continuous Model Description

This section describes the packages used for continuous model description of domains and fields. These are `Boundary`, `Domain`, and `Field`, which are discretization independent information needed for the PDE problem. An overview of the packages in the framework is shown in Figure 1.

The geometry of a domain is described using continuous parametric curves, which is a fairly general representation and is easy to discretize. Each domain object contains a boundary object describing its boundary, i.e., the boundary defines the domain. The direction of the parametric curve representing the boundary decides on which side of the boundary the domain resides. Usually, the domain is on the left of the boundary, i.e., the curve is followed in counter-clockwise direction around a point in the domain.

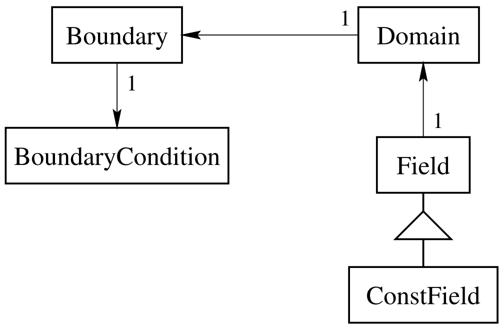

Figure 1: Overview of the packages for continuous domain and field description. ConstField inherits Field, i.e., it is a field with a known value (time-dependent or time-independent). Arrows represent aggregate, e.g. a domain object contains one boundary object, which contains one boundary condition object.

3.1 Boundary Definition

A boundary contains a shape function representing the parametric curve defined for parameters in the range $[0,1]$. The shape function can be seen as a mapping from a real value, the parameter, to the coordinate vector *x*. In two dimensions, one parameter suffices, in three dimensions, two parameters are needed. The specific return value of the shape function has the type `BPoint`, which is a point with additional information about the boundary conditions in that point.

The base package for all boundary types is the package called `Boundary`:

```
package Boundary
  replaceable function shape
    input Real u;
    input Data data;
    output BPoint x;
  end shape;

  replaceable record Data
    parameter BoundaryCondition.Data bc;
  end Data;
end Boundary;
```

The formal parameter `data` to the shape function contains the actual data of the specific boundary object. The record `BPoint`, representing a point with boundary condition information, is defined as follows:

```
type BPoint = Real[3] "x, y and boundary part index";
```

Here, index 1 and 2 represent the coordinates in two-dimensions, while the third value is the boundary condition index, needed by the discretization and solution steps.

3.2 Domain Definition

A base domain type called `Domain` is declared with a boundary instance defining the actual geometry of the domain:

```
package Domain
  replaceable package boundaryP = Boundary
    extends Boundary; // base class restriction

  replaceable record Data
    parameter boundaryP.Data boundary;
  end Data;

  function discretizeBoundary
    input Integer n;
    input boundaryP.Data d;
    output BPoint p[n];
  algorithm
    for i in 1:n loop
      p[i, :] := boundaryP.shape((i - 1)/n, d);
    end for;
  end discretizeBoundary;
end Domain;
```

The restriction specifies that if the package `boundaryP` is replaced, the replacing package must be a subtype of `Boundary`.
The function `discretizeBoundary` must reside in the `Domain` package in order that the correct shape function is called, depending on the replaceable package `boundaryP`. The discretization simply calculates a given number of points uniformly distributed on the boundary. The data record of the domain contains the `boundary` record, which is the actual data record of the selected boundary type.

3.3 Fields

A field represents a mapping from a domain to scalar or vector values. The domain is declared as a replaceable package, which can be replaced by a package extending the `Domain` package described in Section 3.2. The replaceable type `FieldType` determines the value type of the field. The data record contains the data of the domain:

```
package Field
  replaceable type FieldType = Real;
  replaceable package domainP = Domain
    extends Domain; // base class restriction

  replaceable record Data
    parameter domainP.Data domain;
  end Data;

  replaceable function value
    input Point x;
    input Data d;
    output FieldType y;
  algorithm
    y := 0;
  end value;
end Field;
```

The function `value` represents the mapping, which can be defined when specifying fields with known values. Fields with unknown values that must be solved for during simulation may use value functions that interpolate the values for given coordinates.

3.3.1 A Field Example

An example showing a field with time-constant values follows:

```
model FieldExample
  function myfieldfunc
     input Point x;
     input myFieldP.Data d;
     output myFieldP.FieldType y;
  algorithm
     y := cos(2*PI*x[1]/6) + sin(2*PI*x[2]/6);
  end myfield;

  package omegaP =
    Domain (redeclare package boundaryP=Circle);
  package myFieldP =
    Field (redeclare package domainP=omegaP,
           redeclare function value=myfieldfunc);

  parameter Circle.Data bnd(radius=2);
  parameter omegaP.Data omega(boundary=bnd);
  parameter myFieldP.Data myfield(domain=omega);
end FieldExample;
```

The field function `myfieldfunc` defines the mapping from the space coordinates to the field values of type `Real`.

3.4 Included Boundaries

Some predefined boundaries can be found in the package `Boundaries`. All these packages extend the basic package `Boundary`. Therefore the data records in each boundary contain the parameter `bc` of type `BoundaryCondition.Data`, containing the boundary condition information. Boundary conditions are described in Section 4.3. An overview of the included boundaries can be seen in Figure 2. They are also briefly described in the following sections.

3.4.1 Line

Line is a straight line defined by two points, the start and the end points of the line. The data record of `Line` follows:

```
  redeclare record extends Data
    parameter Point p1;
    parameter Point p2;
  end Data;
```

The shape function simply interpolates the points linearly between the end points:

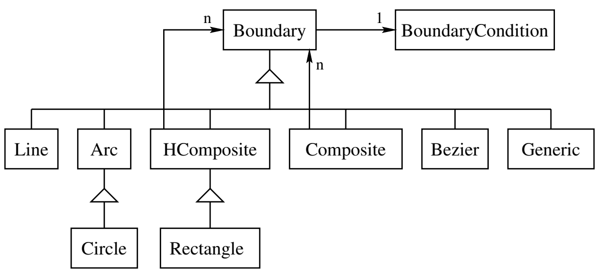

Figure 2: Predefined boundaries contained in the package `Boundaries`. `Composite` is a boundary consisting of boundary parts of different types. `HComposite` (homogeneous composite) consists of boundary parts of the same type. `Generic` is a boundary type that can represent the other concrete boundary types and is used in the `Composite` boundary.

```
redeclare function shape
  input Real u;
  input Data d;
  output BPoint x;
algorithm
  x[1:2] := d.p1 + u*(d.p2 - d.p1);
  x[3] := d.bc.index;
end shape;
```

The boundary condition index is passed through to the points on the boundary.

3.4.2 Arc

An arc is part of a circular boundary with given start and end angles around a center and with a given radius:

```
redeclare record Data
  extends Boundary.Data;
  parameter Point c={0,0};
  parameter Real r=1;
  parameter Real a_start=0;
  parameter Real a_end=2*pi;
end Data;
```

Default values for the parameter gives a full circle. The shape function calculates the position using `sin` and `cos` functions:

```
redeclare function shape
  input Real u;
  input Data d;
  output BPoint x;
protected
  Real a=(d.a_end - d.a_start);
algorithm
  x[1:2] := d.c + d.r*{cos(d.a_start + a*u),
                       sin(d.a_start + a*u)};
  x[3] := d.bc.index;
end shape;
```

3.4.3 Circle

A circle is simply defined by extending `Arc` and giving the angles for a full circle:

```
package Circle
  extends Arc(Data(a_start=0, a_end=2*pi));
end Circle;
```

3.4.4 Rectangle

A rectangle declares four lines as components, with the names `bottom`, `right`, `top` and `left`. For example `bottom` is declared as follows:

```
parameter Line.Data bottom(
  p1=p,
  p2=p + {w,0},
  bc(index=1, name="bottom"));
```

The parameters of the rectangle are `p`, `w` and `h`, representing the bottom left corner, the width and the height, respectively.
The rectangle class extends the `HComposite` package, which is a container for several boundary parts of the same type, as described below. The `Rectangle` package is defined as follows:

```
package Rectangle
  extends HComposite (
    redeclare package PartType = Line);

  redeclare record
    extends Data(bnddata(
      n=4,
      parts={bottom,right,top,left}));
    parameter Line.Data bottom(
      p1=p,
      p2=p + {w,0},
      bc(index=1, name="bottom"));
    // right, top and left defined similarly
  end Data;
end Rectangle;
```

Hence, `bnddata` is a data record inside the rectangle record, with the parts initialized to the vector containing the four declared lines, and as the `PartType` declared as `Line`, accordingly.

3.4.5 Bézier

The `Bézier` boundary package uses a number of control points given as parameters to calculate the coordinates of the points on a bézier curve, using De Casteljau's Algorithm [2]. The data record for `Bezier` package follows:

```
redeclare record extends Data
  parameter Integer n=1;
  parameter Point p[n];
end Data;
```

The shape function implements the algorithm for calculating the coordinates of a point on the curve, given the parameter `u`:

```
redeclare function shape
  input Real u;
  input Data d;
  output BPoint x;
protected
```

```
  Point q[:]=d.p;
algorithm
  for k in 1:(d.n - 1) loop
    for i in 1:(d.n - k) loop
      q[i, :] := (1 - u)*q[i, :] + u*q[i + 1, :];
    end for;
  end for;
  x[1:2] := q[1, :];
  x[3] := d.bc.index;
end shape;
```

3.4.6 Generic

The boundary package Generic is needed in order to define composite boundaries containing boundary parts of different types. Since there are no pointers or union types in Modelica, it is not possible to declare a container for boundary parts where each part can be a subclass of Boundary which is not known at the time of library development. Hence, the Generic package contains an enum parameter deciding the type of the boundary part, and data records for each of the existing types that can be selected. This leads to a lot of overhead, since only one of the records are actually used, but unused parameters are optimized away during the compilation and this does not affect the resulting simulation code. In future implementations, union types or other solutions for polymorphism might allow more efficient implementation of generic boundary types.
The enumeration type and the data record for the Generic boundary type follows:

```
  type PartTypeEnum = enumeration (
      line,
      arc,
      circle,
      rectangle);

  redeclare replaceable record Data
    parameter PartTypeEnum partType;
    parameter Line.Data line;
    parameter Arc.Data arc;
    parameter Circle.Data circle;
    parameter Rectangle.Data rectangle;
  end Data;
```

Because of lack of polymorphism, e.g. virtual functions, the shape function must check the enumeration variable and call the correct shape function:

```
redeclare function shape
  input Real u;
  input Data d;
  output BPoint x;
algorithm
  if d.partType==PartTypeEnum.line then
    x := Line.shape(u, d.line);
  elseif d.partType==PartTypeEnum.arc then
    x := Arc.shape(u, d.arc);
  elseif d.partType==PartTypeEnum.circle then
    x := Circle.shape(u, d.circle);
  elseif d.partType==PartTypeEnum.rectangle then
    x := Rectangle.shape(u, d.rectangle);
  end if;
end shape;
```

3.4.7 Composite

The Composite boundary simply uses a given number of Generic boundaries to build a complete boundary using parts of different types:

```
package PartType = Boundaries.Generic;

redeclare replaceable record extends Data
  parameter Integer n=1;
  parameter PartType.Data parts[n];
end Data;
```

The shape function simply calls the shape function in the Generic boundary package, using the index calculated by dividing the formal parameter u uniformly among the existing parts:

```
redeclare function shape
  input Real u;
  input Data d;
  output BPoint x;
protected
  Real s=d.n*u;
  Integer is=integer(s);
algorithm
  x := PartType.shape(s - is, d.parts[1 + is]);
end shape;
```

Here, is contains the part index corresponding to the value of the formal parameter u, and s-is is the new parameter value scaled to map to the parameter range of that particular boundary part. For example, if the shape function is called for a boundary containing four parts with $u = 0.8$, the value of *is* will be $integer(4*0.8) = 3$ and the value of $s - is$ will be $4*0.8-3 = 0.2$, mapping to the u value on the fourth boundary part.

HComposite is a simplified version of the Composite boundary, containing only parts of the same type.

4 Equation Models

The Equation models contain all the different components of the PDE model, and handle the spatial discretization and the declaration of the discrete model equations. The continuous components of the model, i.e., the domain, its boundary, the boundary conditions and the field, are declared here. Their discrete counterparts are declared and initialized automatically from the continuous components, using given discretization parameters. The spatial discretization is done by calling the finite element solver, which can be implemented in Modelica, or an external solver called through the Modelica external function interface. The declared equations use the spatially discretized model.

4.1 The Poisson Equation

The Poisson equation is a simple example of a stationary (time-independent) model. In differential form, the equation is

$$-\nabla \cdot (c\nabla u) = f \qquad \text{in} \quad \Omega \tag{1}$$

where u is the unknown field, c is a space-dependent coefficient, f is the source term and Ω is the domain.

4.2 The Diffusion Equation

The diffusion equation for a field u is:

$$\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) = f \qquad \text{in} \quad \Omega \tag{2}$$

where c is a space-dependent coefficient, f is the source term and Ω is the domain.

4.3 Boundary conditions

In both cases the boundary conditions may be Dirichlet, Neumann or mixed. The Diriclet boundary conditions is used where the value of the unknown field is known on the boundary:

$$u = g \qquad \text{on} \quad \Omega \tag{3}$$

The Neumann boundary condition is used when the value of the normal derivative of the field is known on the boundary:

$$\frac{\partial u}{\partial n} = g \qquad \text{on} \quad \Omega \tag{4}$$

The mixed boundary condition, also called the Robin boundary condition, contains both the value of the field and the normal derivative:

$$a\frac{\partial u}{\partial n} + bu = g \qquad \text{on} \quad \Omega \tag{5}$$

5 Discretization

So far only the continuous parts of the packages have been discussed. These are independent of the discretization, and thus also the solution method, e.g. the finite element method or the finite difference method. The method for discretization of the domain depends on which solution method is used. The finite element package is described in the following section. Packages for the finite difference method exist for an earlier prototype of the framework. Also, packages for the finite volume method are being considered.

5.1 The Finite Element Package

For the finite element solver, the domain is represented by a triangular mesh. The mesh generator used in this work requires a polygon describing the boundary of the domain as input. This polygon is generated by discretizing the domain boundary using the shape function. A simple discretization function sampling a given number of points uniformly on the boundary is implemented as follows:

```
function discretizeBoundary
  input Integer n;
  input boundaryP.Data d;
  output BPoint p[n];
algorithm
  for i in 1:n loop
    p[i,:] := boundaryP.shape((i-1)/n, d);
  end for;
end discretizeBoundary;
```

The resulting polygon is given to the mesh generator bamg [1]. The triangulation is then imported to Modelica. Figure 3 shows the overview of the packages used in this process.

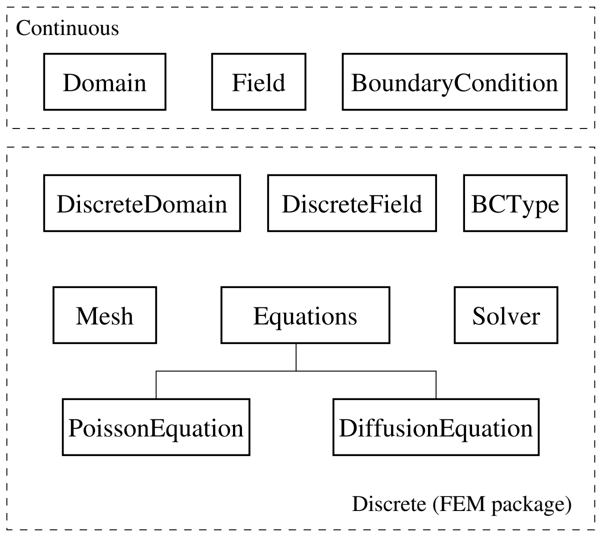

Figure 3: Packages involved in the discretization using the finite element method. The user has only to deal with the continuous part when using the equation packages.

The complete discretization and solution process is depicted in Figure 4. The external stiffness matrix calculation can be exchanged with internal code, i.e., functions implemented in Modelica. A prototype implementation in Modelica exists for discretization of the Poisson equation with homogeneous Dirichlet boundary conditions.

5.1.1 DiscreteDomain

`DiscreteDomain` is the discrete version of `Domain`. It contains a replaceable package `domainP`, representing the continuous version of the domain.

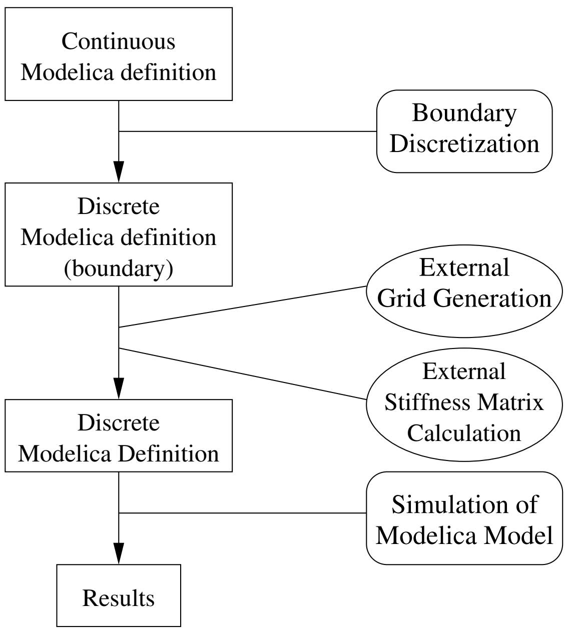

Figure 4: Solution diagram. The boxes on the left show the data flow. The rounded boxes show tools implemented in Modelica, and ellipses show external tools.

The discretization is done automatically, once `DiscreteDomain` is declared with a given `Domain` package. `DiscreteDomain` is defined as follows:

```
package DiscreteDomain
 replaceable package domainP = Domain
   extends Domain; // base class restriction

 replaceable record Data
   parameter Integer nbp;
   parameter domainP.Data domain;
   // A parameter to the mesh generator
   // specifying detail level, lesser means
   // more triangles
   parameter Real refine=0.7;

   // Array of discrete points on the boundary
   parameter BPoint boundary[nbp]=
     domainP.discretizeBoundary(nbp,
                                domain.boundary);
   parameter Mesh.Data mesh(
     n=size(boundary, 1),
     polygon=boundary[:, 1:2],
     bc=integer(boundary[:, 3]),
     refine=refine);
   parameter Integer boundarySize=
     size(boundary, 1);
 end Data;
end DiscreteDomain;
```

The actual mesh generation is done when the `mesh` component is instantiated by the compiler, i.e., the `Mesh` package contains the actual calls to the external mesh generator.

5.1.2 DiscreteField

The package `DiscreteField` is incapsulates the conversion of a continuous field to a discrete field, using a given discrete domain. A discrete field contains two separate arrays of discrete points in the domain, one array containing the unknown values and one containing the known values, e.g. from given boundary conditions. This representation corresponds to the representation used in Rheolef [7], in order to simplify the solver interface. Both arrays are indirect, e.g. they contain indices of the actual points in the mesh representation. The `DiscreteField` package is defined as follows:

```
package DiscreteField
  replaceable package fieldP = Field;
  replaceable package ddomainP = DiscreteDomain;

  replaceable record Data
    parameter ddomainP.Data ddomain;
    parameter fieldP.Data field;
    parameter FEMSolver.FormSize formsize;
    parameter Integer u_indices[formsize.nu];
    parameter Integer b_indices[formsize.nb];
    fieldP.FieldType val_u[formsize.nu](
      start=zeros(formsize.nu));
    fieldP.FieldType val_b[formsize.nb];

    parameter Integer fieldSize_u=size(val_u, 1);
    parameter Integer fieldSize_b=size(val_b, 1);
  end Data;

end DiscreteField;
```

Here, the default start values for the unknowns are set to zeros. This value is overridden in the discrete parts of the equation models, for appropriate initial value setting. `FormSize` contains the sizes of the two arrays of discrete values, and is imported from the external solver since the sizes depend on the boundary conditions actually used in the model. Basically, Dirichlet and mixed boundary conditions decides the number of known variables.

5.1.3 Equation Discretization

The spatial derivatives in the equations are discretized using the external solver Rheolef [7], which is automatically called from the equation models through external functions. Rheolef performs the assembling of the matrix needed for the space discrete DAE system. The result of the assembly is a coefficient matrix for the unknown field values at the discrete points of the domain. The resulting matrices are imported to Modelica through external functions and used in the actual equations in the equation models. The final, possibly time-dependent, equation system, is simulated in Dymola. An example solved using this framework is shown in the following section.

6 Example

The result of the discretization of the equation, i.e., the assembly step, is a coefficient matrix for the unknown field values at the discrete points of the domain. The discrete part can be completely handled by the equation model, hiding the details from the user, as shown in the example using the PoissonEquation model:

```
model GenericBoundaryPoissonExample
  import PDEbhjl.Boundaries.*;
  import PDEbhjl.*;

  parameter Integer n=40;
  parameter Real refine=0.5;
  parameter Point p0={1,1};
  parameter Real w=5;
  parameter Real h=3;
  parameter Real r=0.5;
  parameter Real cw=5;

  package myBoundaryP = MyGenericBoundary;

  parameter myBoundaryP.Data mybnd(
    p0=p0,
    w=w,
    h=h,
    cw=cw,
    bottom(bc=dirzero),
    right(bc=dirfive),
    top(bc=dirzero),
    left(bc=dirfive));

  package omegaP = Domain(
    redeclare package boundaryP=myBoundaryP);
  parameter omegaP.Data omega(boundary=mybnd);

  parameter BoundaryCondition.Data dirzero(
    bcType=BoundaryCondition.dirichlet,
    g=0,
    q=0,
    index=1,
    name="dirzero");

  parameter BoundaryCondition.Data dirfive(
    bcType=BoundaryCondition.neumann,
    g=5,
    q=1,
    index=2,
    name="dirfive");

  parameter BoundaryCondition.Data bclist[:] =
    { dirzero,
      dirfive };

  package PDE =
    PDEbhjl.FEMForms.Equations.Poisson2D
      (redeclare package domainP = omegaP);

  PDE.Equation pde(
    domain=omega,
    nbp=n,
    refine=refine,
    g0=1,
    nbc=size(bclist, 1),
    bc=bclist);
end GenericBoundaryPoissonExample;
```

Here, two different boundary conditions are assigned to different parts of the boundary. The boundary used here is defined as follows:

```
package MyGenericBoundary
  extends Boundary;

  redeclare record extends Data
    parameter Point p0;
    parameter Real w;
    parameter Real h;
    parameter Real cw;

    parameter Real ch=h;
    parameter Point cc=p0 + {w,h/2};

    parameter Line.Data bottom(
      p1=p0,
      p2=p0 + {w,0});
    parameter Line.Data top(
      p1=p0 + {w,h},
      p2=p0 + {0,h});
    parameter Line.Data left(
      p1=p0 + {0,h},
      p2=p0);

    parameter Bezier.Data right(
      n=8,
      p=fill(cc, 8) +
       { {0.0,-0.5},{0.0,-0.2},{0.0,0.0},
         {-0.85,-0.85},{-0.85,0.85},{0.0,0.0},
         {0.0,0.2}, {0.0,0.5}
       } * { {cw,0}, {0,ch} });

    parameter Composite.Data boundary(
      parts1(line=bottom,
             partType=PartTypeEnumC.line),
      parts2(bezier=right,
             partType=PartTypeEnumC.bezier),
      parts3(line=top,
             partType=PartTypeEnumC.line),
      parts4(line=left,
             partType=PartTypeEnumC.line));
  end Data;

  redeclare function shape
    input Real u;
    input Data d;
    output BPoint x;
  algorithm
    x := Composite.shape(u, d.boundary);
  end shape;

end MyGenericBoundary;
```

The basic contents of the `Poisson2D` equation model used above is defined as follows:

```
package Poisson2D "Poisson problem 2D"
    package uDFieldP = DiscreteField(
      redeclare package ddomainP = ddomainP,
      redeclare package fieldP = uFieldP);

    uDFieldP.Data fd(
      ddomain=ddomain,
      field=uField,
      formsize=formsize,
      u_indices=u_indices,
      b_indices=b_indices,
      val_u(start={1 for i in 1:formsize.nu}));
  equation
    laplace_uu * fd.val_u
      = mass_uu*g_rhs.val_u + mass_ub*g_rhs.val_b
          - laplace_ub*fd.val_b;
    fd.val_b = bvals; // known boundary values
  end Equation;
end Poisson2D;
```

The matrices `laplace_uu`, `mass_uu`, `mass_ub` and `laplace_ub` are retrieved from the external solver Rheolef. Also `bvals` is calculated by the external solver. For diffusion problems, additional matrices are retrieved for the coefficients for the time derivatives of the unknowns.

The plot of the simulation result can be seen in Figure 5. For comparison, same model is exported to and solved in FEMLAB. Figure 6 shows the result generated by FEMLAB. The triangulation of the domain in both cases can be seen in Figure 7.

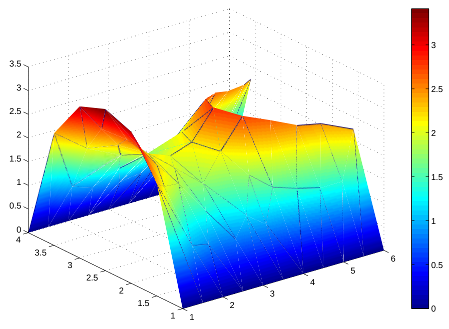

Figure 5: Results from solving the Poisson equation (steady-state) in Dymola.

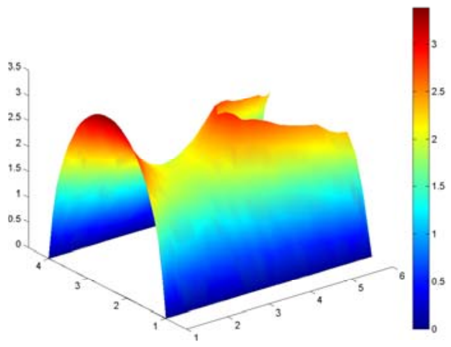

Figure 6: Results from solving the Poisson equation (steady-state) in FEMLAB.

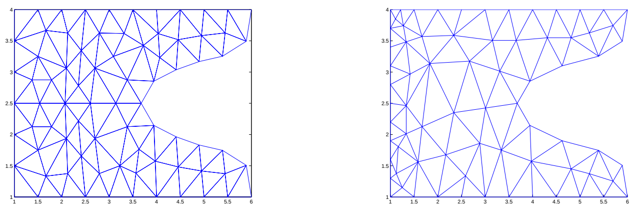

Figure 7: The meshes automatically generated from domain description in Modelica. Bamg mesh on the left, FEMLAB solution mesh on the right.

7 Conclusion and Future Work

The packages presented here give a general framework for easily defining general domains over which the predefined PDE models from the framework can be solved. New boundaries are easy to define using the existing boundaries as components, as shown in Section 6. Additional standard boundaries can also be added to the `Boundaries` package for future use.

New PDE models are also easy to add to the framework. Models that can be formulated using forms as described in the Rheolef User Manual [8] can be added to the framework by using the external function interface and implementing necessary extensions.

Further work is needed on the finite difference and the finite volume packages and adapt them to the current continuous definition framework. Also, the finite difference solver can be improved to support non-rectangular domains.

A simple extension of the framework is to include domains that use the existing standard boundaries. For example, a `CircularDomain` can be defined in the framework as follows:

```
package CircularDomain
  extends Domain(
    redeclare package boundaryP = Circle);
end CircularDomain;
```

Such a domain can be used directly when defining new problems, instead of declaring a general domain each time and replacing the boundary manually.

8 Acknowledgments

This work has been supported by Swedish Foundation for Strategic Research (SSF), in the VISIMOD project.

References

[1] BAMG home page. `http://www-rocq.inria.fr/gamma/cdrom/www/bamg/eng.htm`.

[2] Gerald Farin and Dianne Hansford. *The Essentials of CAGD*. A K Peters, 2000.

[3] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2003. `http://www.mathcore.com/drmodelica/`.

[4] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems*

Modeling - Language Specification Version 2.1, Jan 2004. `http://www.modelica.org/`.

[5] L. Saldamli and P. Fritzson. Field Type and Field Constructor in Modelica. In *Proc. of SIMS 2004, the 45th Conference on Simulation and Modelling*, Copenhagen, Denmark, September 2004.

[6] Levon Saldamli. *PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations*. Linköping Studies in Science and Technology, Licentiate Thesis No. 990, December 2002.

[7] Pierre Saramito, Nicolas Roquet, and Jocelyn Etienne. Rheolef home page. `http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/`, 2002.

[8] Pierre Saramito, Nicolas Roquet, and Jocelyn Etienne. *Rheolef users manual*. 2002. `http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/usrman.ps.gz`.