Domains and Partial Differential Equations in Modelica

Levon Saldamli and Peter Fritzson
Department of Computer and Information Science
Linköping University
Linköping, Sweden

Abstract

Modelica™ is an equation-based object-oriented modeling language that supports models containing ordinary differential equations and differential and algebraic equations. In this article, we make an object-oriented design for extending Modelica with partial differential equations (PDEs) in order to describe and solve initial and boundary-value problems. We present constructs for geometric description of domains and domain boundaries using parametric expressions, and a hierarchical specification of PDEs and boundary conditions using inheritance, with a general PDE as a base model and more specific, application oriented sub-models. Using instances of PDE models, boundary conditions and domains we specify a complete PDE problem. Two environments used for prototype implementations are also described.

1 Introduction

The modeling language Modelica [3, 4, 6, 9] is currently used for modeling and simulation of systems with ordinary differential equations containing time-dependent variables. However, models with variables that vary with location in space and contain partial differential equations (PDEs) are used widely in modeling and simulation. Therefore, there is a need to extend Modelica to support such models.

Solving a PDE problem means finding a function of space variables, and time in time-dependent problems, that is implicitly defined by a PDE system. In order to find a unique solution, the domain of the problem, i.e. the region where the unknown function is to be studied, must be defined. Additionally, some conditions for the boundary of this region must be given. There can be different boundary conditions for different parts of the boundary, and the conditions can be known values of the unknown function or its derivatives. In time-dependent problems, usually some initial conditions must also be given that may involve values of the unknown function or its derivatives at some time point $t = t_0$.

Modelica is an object-oriented language, supporting inheritance and component-based modeling. Extensions to support PDEs should be done with these concepts in mind in order for a PDE problem to be specified in a convenient way similar to other models written in Modelica. Previously, some basic extensions needed in Modelica were presented [13]. The domains were described by defining the limits of the space variables using constants or expressions containing other space variables in order to allow fairly general domains. In this paper, we support a more convenient domain definition, using

parametric expressions for describing the boundary of the domain. We also describe how a problem can be specified with the PDE, the boundary conditions, the domain and its boundary defined as components.

This paper is organized as follows: Section 2 contains an overview of related work, Section 3 presents the problem specification and new language syntax, Section 4 describes the implementation environments and Section 5 contains some conclusions and future work.

2 Related Work

There are different categories of packages for solving PDEs. Some of them are code libraries, where the PDE is not separately specified but a numerical solver is written using a programming language and components from these libraries in order to solve the specific PDE problem. PETsc [1], Diffpack [2] and Overture [11] are some packages in this category. Compose [15] is a framework with an object-oriented design that separates the equation definitions and numerical solver implementation, where equations are defined using the C++ classes in the framework or by adding new classes to define new equations and numerical solvers.

There are also problem solving environments (PSEs), that contain tools for the different steps of the modeling and simulation process, such as graphical tools for defining the domain, tools for specifying or selecting a numerical solver among several solvers, and visualization of the simulation results. PELLPACK [7] is such a problem solving environment that contains several PDE solvers and has a high level language for the PDE problem definition. FEMLAB [5] is another simulation tool, written as a package for MATLAB, with graphical user interface where the user can choose a model among many predefined PDE models, modify its parameters, graphically define the problem domain and assign boundary conditions, simulate the model and visualize the results.

An environment that is more language oriented, analogous to Modelica, is gPROMS [10]. This environment has a high level language for specifying PDE models on rectangular domains, where complex partial differential and algebraic equations and mixed systems of integral, partial and ordinary differential and algebraic equations can be solved.

The approach taken in our present work to extend Modelica with PDEs, called PDE-Modelica, combines the usage of a high level language, object-oriented and component-based modeling, and the possibility to use different solvers and automatic solver generation for a given PDE problem.

3 Domain and PDE definition

In this section, we describe how to define the problem domain using parametric curves. Also, a hierarchical PDE model definition using coefficient-based PDEs similar to FEMLAB's coefficient form is described.

3.1 Curve-based Domain Description

The domain of the PDE problem is $D \subset \mathbb{R}^n$. In this work we consider the two-dimensional case, $n = 2$. In most practical cases it is sufficient to define the domain by a parametric curve $\{(x_s, y_s) \mid s \in [s_{start}, s_{end}]\}$ describing the boundary of the region, which is a quite general way of stating the geometry of the domain. The curve

should be closed and non self-intersecting for the parameter range specified, and the direction of the curve is defined by varying the parameter from its start value to its end value. In the two-dimensional case, the XY-plane is divided into two regions by the curve, one region inside the closed curve and one region outside. The intended domain is defined by the curve together with its direction.

The boundary defined this way is used to generate a mesh to be used by the numerical PDE solver. An external mesh generator is used to generate the mesh.

3.1.1 Single boundary

A domain class can be defined using a new restricted Modelica class called **domain**, where the space variables to be used are declared using the **space** keyword. There are several alternative ways to specify the boundary curve using an expression. Using the **where...in...** construct which already has been used to specify domains for expressions [13], the curve can be defined as follows:

```
domain Cartesian2D "Base class for two dimensional domains"
  space Real x,y;
end Cartesian2D;

domain Circle2D "Circular domain with r=1"
  extends Cartesian2D;
algorithm
  (x,y) := ( cos(2*PI*u), sin(2*PI*u) ) where u in (0,1);
end Circle2D;
```

The boundary of this domain is defined by the curve generated by varying the value of the temporary variable u from 0 to 1. In order for the curve to be closed, the resulting points in the XY-plane at $u = 0$ and $u = 1$ should have the same coordinates. Other requirements might be needed for the curve depending on the mesh generator used by the numerical solution stage. Another alternative is to write the expression in a Modelica function that takes one argument of type **Real**, and returns a tuple of two space variables with type **Real**, and replace the expression above with a call to the function. This way the expression describing the boundary curve can be reused in other domains if needed.

3.1.2 Composite boundary

In many cases, the boundary of the problem domain is not a curve that can be defined by a single expression or function, but rather a number of curves attached together. Also, the boundary conditions for the PDE problem are often different on different parts of the boundary. Therefore, when the boundary curve is specified, there must be a way to refer to different parts of the curve when assigning boundary conditions. One solution to these problems is to have a boundary description that consists of several components, each of which are curves. The boundary components can be declared in the declaration part of the domain description. For example, a rectangular boundary can be defined using four line segments. These parts of the boundary can be instantiated in the declaration part of the domain class as follows:

```
domain Line2D "A line segment"
  extends Cartesian2D;
  parameter Real x0=0, y0=0, x1=1, y1=1;
algorithm
```

```
    (x,y) := ( x0+r*(x1-x0), y0+r*(y1-y0) ) where r in (0,1);
  end Line2D;

  domain Rectangle2D "A 6 by 4 rectangle"
    extends Cartesian2D;
    parameter Real cx=0, cy=0, w=3, h=2;
    Line2D right(x0=cx+w, y0=cy-h, x1=cx+w ,y1=cy+h);
    Line2D top(x0=cx+w, y0=cy+h, x1=cx-w, y1=cy+h);
    Line2D left(x0=cx-w, y0=cy+h, x1=cx-w, y1=cy-h);
    Line2D bottom(x0=cx-w, y0=cy-h, x1=cx+w, y1=cy-h);
  algorithm
    (x,y) := composite(right, top, left, bottom);
  end Rectangle2D;
```

The domain `Rectangle2D` can be seen in Figure 1. The **composite** operator is used to combine several curve segments into a complete boundary. The setting of the start and end points of the line segments and the order of the arguments to the **composite** operator must be consistent, and the direction of the resulting curve must be correct in order that the correct region is defined. Some of these requirements can be automatically fulfilled if the **composite** operator is allowed to translate each given curve segment so that the starting point of that curve matches the end point of the previous curve segment.

Although both `Line2D` and `Rectangle2D` are defined here as domains, they represent different kind of objects. The `Line2D` domain doesn't have a closed boundary, and thus cannot be used as a domain by itself. This difference could be expressed in the definition by for example using the **partial** keyword in the definition of `Line2D`:

```
  partial domain Line2D "Defines a part of a boundary"
    ...
```

Another alternative is to use a different keyword than **domain** for classes that represent only parts of a boundary.

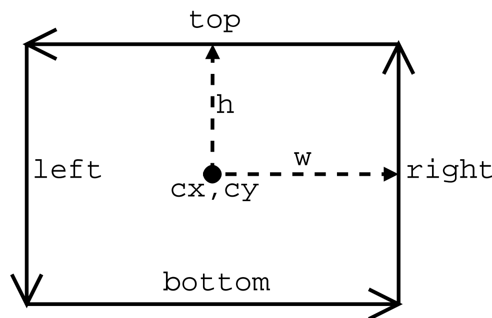

Figure 1: A rectangular domain `Rectangle2D`, defined using line segments. Note that the direction of the lines must be consistent.

3.2 Hierarchical Definition of PDEs and Boundary Conditions

In order to simplify PDE model definition, a general PDE model can be written as a base model in PDE-Modelica with the coefficients as parameters. This model can

either be instantiated directly with appropriate modifications to the parameters or used as a base class to define a more specific PDE model with some parameters set which subsequently can be instantiated and used when needed. Similarly, boundary conditions can be defined using base models and inheritance. As an example, we first define a new type for variables of type **Real** with two space dimensions by modifying the **defdomain** attribute of **Real**, as introduced in previous work[1] [13], and then define a base PDE model:

```
type Real2D = Real(defdomain=Cartesian2D);

model PDE2D
  Real2D u;
end PDE2D;

model PDECoeff2D
  extends PDE2D;
  parameter Real2D da = 0;
  parameter Real2D c = 0;
  parameter Real2D alfa = 0;
  parameter Real2D gamma = 0;
  parameter Real2D a = 0;
  parameter Real2D beta = 0;
  parameter Real2D f = 0;
equation
  da*der(u)-div(c*grad(u)+alfa*u-gamma)+a*u+beta*grad(u)=f;
end PDECoeff2D;
```

The variable u represents the unknown variable, a function of time and the space variables, declared using the derived type **Real2D**. All parameters can be constants or functions of the space variables. The default values of the parameters are set to zero. The **der** operator is an operator in current Modelica and defines the first time-derivative of a variable. The **div** and **grad** operators can be additional operators in PDE-Modelica corresponding to the partial differential operators **divergence** and **gradient** that are often used in mathematical literature. The equation above written with mathematical notation follows:

$$d_a \frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u + \alpha u - \gamma) + au + \beta \nabla u = f$$

Using `PDECoeff2D` as the base model, a simple heat transfer model can now be written as:

```
model HeatTransfer
  extends PDECoeff2D(da=1, c=1);
end HeatTransfer;
```

A more general heat transfer model would contain other coefficients that are used in heat transfer problems instead, and equations to map those to the coefficients of the `PDECoeff2D` model.

A Neumann boundary condition for a heat equation can be written by first writing a generalized Neumann boundary condition:

[1] The attribute introduced previously was called **domain**, but this keyword is already used as the restricted class name for defining domains. The keyword **defdomain** was chosen as a short word for "definition domain".

```
model GenNeumann "Generalized Neumann boundary condition"
  extends PDE2D;
  parameter Real c = 0;
  parameter Real alfa = 0;
  parameter Real gamma = 0;
  parameter Real q = 0;
  parameter Real g = 0;
equation
  n*(c*grad2(u)+alfa*u-gamma) + q*u = g;
end GenNeumann;
```

In mathematical notation, this equation would be:

$$n \cdot (c\nabla u + \alpha u - \gamma) + qu = g$$

The variable `n` is special and must be handled separately, because it represents the normal vector of the boundary of the domain. It is needed because the normal derivative of the unknown `u` is used in this boundary condition. A better operator to use can be `nder`, which can represent the normal derivative of a variable, with respect to the associated domain.

A more specific model of the Neumann boundary condition for heat transfer problems can now be defined by inheriting `GenNeumann`, and modifying the parameters in the **extends** statement:

```
model HeatNeumann "For heat transfer problems"
  extends GenNeumann(c=k, q=hh, g = qh+hh*Tinf);
  parameter Real k=1;
  parameter Real qh=0;
  parameter Real hh=1;
  parameter Real Tinf=25;
end HeatNeumann;
```

3.3 Problem definition

Once the models for the PDE and the boundary conditions are written and the domain is defined, the problem can be put together by instantiating the models and the domain and assigning boundary conditions to the domain boundary. For example:

```
model PDEModel
  parameter Real2D u_init = 0; // initial condition
  Real2D u(start=u_init);
  HeatNeumann h_iso;
  HeatNeumann h_glass(qh=1.5);
  HeatTransfer ht;
  Rectangle2D dom(eq=ht,
                  left(eq=h_glass),
                  right(eq=h_iso),
                  top(eq=h_iso),
                  bottom(eq=h_iso));
end PDEModel;
```

Here, the boundary condition `HeatNeumann` is instantiated twice, `h_iso` with default values, and `h_glass` with a different value for the heat transfer coefficient. The PDE is called `ht`, and the domain is a rectangular domain declared as `dom`.

The alternative used here for assigning boundary conditions to different parts of the boundary is to use a variable of type PDE2D in the domain classes and assign the boundary condition models or the PDE model during the declaration of the domain. This variable can be added to the base domain class Cartesian2D, as follows:

```
domain Cartesian2D
  outer space Real x,y;
  PDE2D eq;
end Cartesian2D;
```

Another alternative way of associating boundary conditions and parts of the boundary is to use a special operator, for example **pde_domain**:

```
model PDEModel
  parameter Real u_init = 0;
  inner Real2D u(start=u_init);
  HeatNeumann h_iso;
  HeatNeumann h_glass(qh=1.5);
  HeatTransfer ht;
  Rectangle2D dom;
equation
  pde_domain(ht, dom);
  pde_domain(h_glass, dom.left);
  pde_domain(h_iso, dom.right, dom.top, dom.bottom);
end PDEModel;
```

4 Implementation

We are working with two prototype environments where the ideas described in Section 3 are being tested. The prototype written in Mathematica uses MathModelica [8] as the Modelica implementation and a numerical PDE solver generator [14] for solving the PDEs. The different modules of this environment can be seen in Figure 2. Here, the models are written in MathModelica syntax, and the domain analyzer generates domain information that is sent to an external mesh generator. The PDE analyzer collects the PDEs and the boundary conditions and generates the equations for the problem. These equations are sent to the solver generator that generates C++ code for solving the PDE with the finite element method. The generated solver is not dependent of the domain, i.e. different domains and meshes can be used with the same generated solver. The advantages of this environment is the access to symbolic manipulation in Mathematica, and the MathModelica input format that is easy to extend in order to test new language syntax extensions.

The other prototype environment consists of a Modelica parser, a compiler called modeq generated by RML [12] from a semantic description of Modelica using natural semantics, an external mesh generator and a PDE solver. An overview of this environment can be seen in Figure 3.

The current version of the prototype ignores the equation parts of the PDE and boundary condition models and assumes a certain structure of the PDE. A specific solver adapted to the problem is called automatically with the parameters extracted from the models. This can be done because the base model approach is used when writing the PDE models, i.e. the solver needs only to be associated with the base model, and parameters of the base model are transferred to the solver.

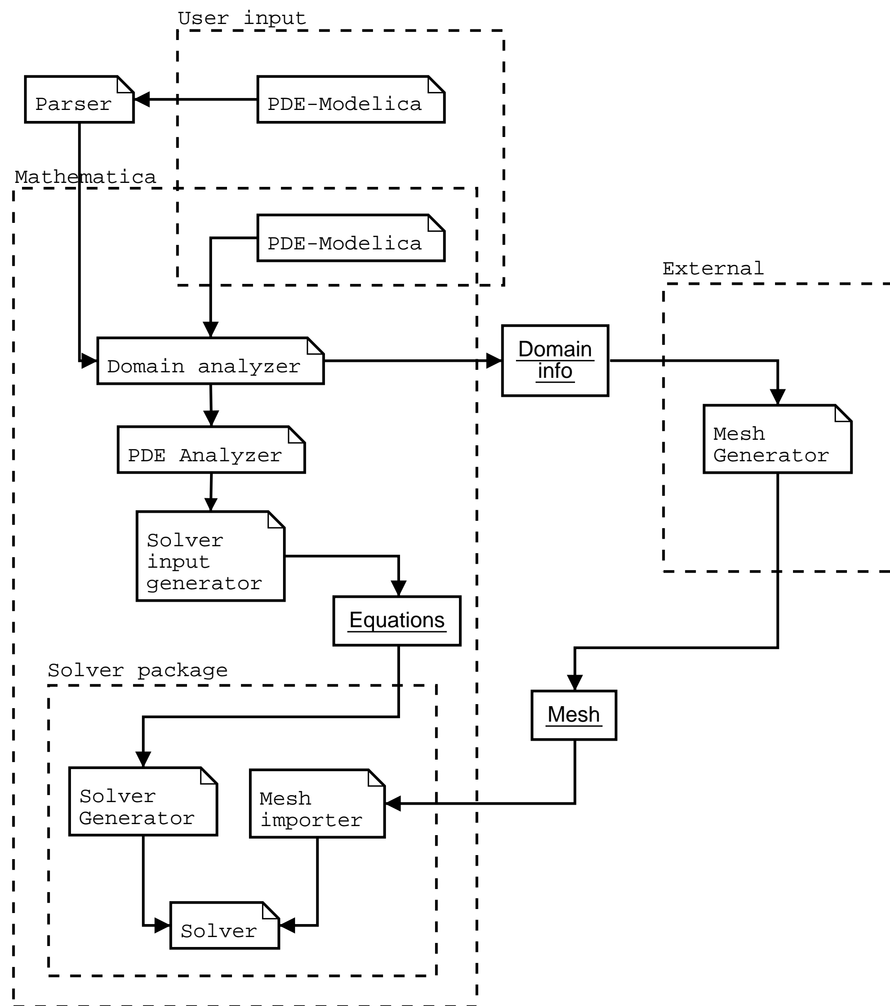

Figure 2: The PDE-Modelica prototype in the Mathematica environment

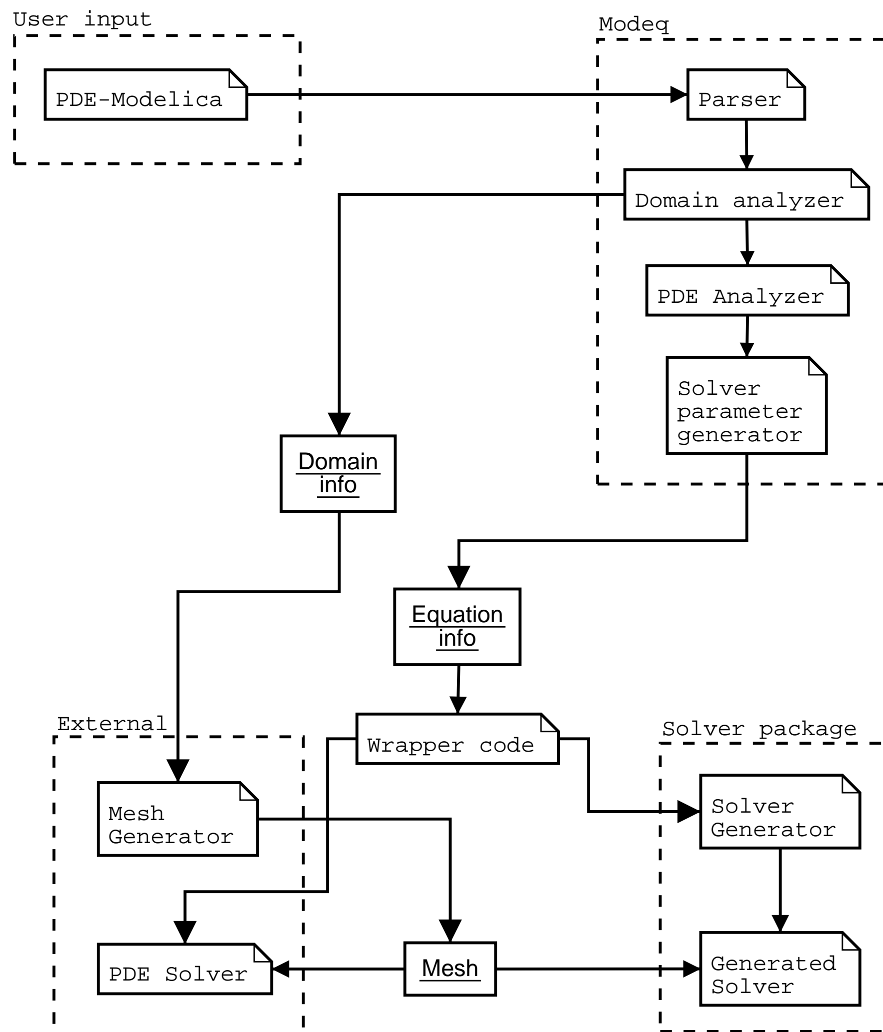

Figure 3: The PDE-Modelica prototype using the Modelica translator called `modeq` that is generated from natural semantics specification of Modelica in RML.

The implementation work is ongoing and we plan to run some first examples using a FEM solver in a few weeks. Using Modeq, we can extract the model parameters from the shown examples, generate functions for single domain boundaries, discretize it and call an external mesh generator. Adding a solver and test the prototype remains to be done. Also, the FEM solver generator for the Mathematica environment is being developed. We have previously solved some example models using the finite difference solver generator [13].

5 Conclusions and Future Work

We have presented a design for specifying PDE problem domains in Modelica by expressing the boundary of the domain as a single curve or a list of curves. We have also shown a simple, hierarchical example of a PDE model and boundary condition models and how these can be put together in a problem specification together with a domain.

Our future work will consist of adding support for the equation parts of the PDE and boundary condition models, instead of having predefined equations. Also, modeling with both PDE models and the current Modelica models with DAEs and the interaction between these models needs to be considered. The combination of domains using set operations, and composition of domains into bigger domains using connect statements with different PDE models on each partial domain is another extension to consider.

References

[1] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc/, 2001.

[2] Diffpack home page. http://www.diffpack.com/.

[3] H. Elmqvist, S. E. Mattsson, and M. Otter. A language for physical system modeling, visualization and interaction. In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design*, Hawaii, Aug. 1999.

[4] H. Elmqvist and S.E. Mattsson. Modelica – the next generation modeling language – an international design effort. In *Proceedings of the First World Congress on System Simulation*, Singapore, Sept. 1–3 1997.

[5] FEMLAB home page. http://www.femlab.com/.

[6] P. Fritzson and V. Engelson. Modelica—A unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1998.

[7] E. N. Houstis, J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms. *ACM Transactions on Mathematical Software*, 24(1):30–73, March 1998.

[8] M. Jirstrand. MathModelica, a Full System Simulation Tool. In P. Fritzson, editor, *Proc. of the Modelica Workshop 2000*, Lund University, Lund, Sweden, October 2000.

[9] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 1.4*, Dec 2000. http://www.modelica.org.

[10] M. Oh. *Modelling and Simulation of Combined Lumped and Distributed Processes.* PhD thesis, University of London, 1995.

[11] Overture home page. http://www.llnl.gov/CASC/Overture/.

[12] M. Pettersson. Compiling Natural Semantics. volume 1549 of *LNCS*. Springer-Verlag, 1999.

[13] L. Saldamli and P. Fritzson. A Modelica-based Language for Object-Oriented Modeling with Partial Differential Equations. In A. Heemink, L. Dekker, H. de Swaan Arons, I. Smith, and T. van Stijn, editors, *Proc. of the 4th International EUROSIM Congress*, Delft, The Netherlands, June 2001.

[14] K. Sheshadri and P. Fritzson. A Mathematica-based PDE-solver generator. In *Proc. of the Scandinavian Simulation Society (SIMS) Conference*, Linköping University, Linköping, Sweden, September 1999.

[15] K. Åhlander. *An Object-Oriented Framework for PDE Solvers.* PhD thesis, Uppsala University, 1999.