Extending Modelica for Partial Differential Equations

Levon Saldamli[*] , Peter Fritzson[*] and Bernhard Bachmann[†]

Abstract

Currently, Modelica only supports models containing constants, time-dependent variables, and time-derivatives of variables, i.e. ordinary differential and algebraic equations. In this article, we present how the Modelica language can be extended to support object-oriented modeling with partial differential equations (PDEs), in order to solve initial and boundary value problems. The techniques we present have fairly general applicability to 1D, 2D or 3D domains, although we focus mostly on 2D domains in this paper. We also describe the architecture of a prototype implementation where the PDE problem is translated and passed to an external mesh generator and a PDE solver for solution using the finite element method. An example of a stationary heat conduction problem is included together with execution results.

1 Introduction

The modeling language Modelica [4, 5, 7, 10] is currently used for modeling and simulation of systems with ordinary differential equations containing time-dependent variables and derivatives of such variables with respect to time. It is desirable to also specify models where variables vary with position in space and where partial differential equations (PDEs) occur. Therefore, there is a need to extend Modelica to support such models.

A PDE problem is solved in order to find an unknown, spatially distributed function u, that is implicitly defined by a partial differential equation. For a unique solution boundary conditions at the boundary of the geometric region of the problem is needed, and also the initial conditions if the problem is time-dependent. 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. Initial conditions can consist of values of the unknown function or its derivatives.

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 [14]. 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, Section 5 illustrates an example problem and its solution, and Section 6 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 [2], Diffpack [3] and Overture [12] are some packages in this category. Compose [17] is a similar package, written as a framework built upon Overture, with an object-oriented design that separates the equation definitions and numerical solver implemen-

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

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

tation. Equations in Compose 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, that contain integrated 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 tools for visualization of the simulation results. PELLPACK [9] is such a problem solving environment that contains several PDE solvers and has a high level language for the PDE problem definition. FEMLAB [6] which is a package for MATLAB, is another simulation tool, 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 [11]. 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 lines and parametric curves. Also, a hierarchical PDE model definition using coefficient-based PDEs similar to FEMLAB's coefficient form is described.

3.1 Domain Description

The domain of the PDE problem is $D \subset \mathbb{R}^n$. In this paper 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 sufficiently general way of stating the geometry of the domain. The curve should be closed and non self-intersecting for the parameter range specified. In the two-dimensional case, the XY-plane is divided into two regions by the curve, with the intended domain being the region at the left side of the curve.

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

From the boundary definition, an external mesh generator is called to generate a triangular mesh which is passed to the numerical solver.

A domain class is defined by introducing a new kind of restricted class in Modelica called `domain`, where the independent space variables to be used are declared using the `space` keyword, and the boundary is described in a special section called `boundary`. The `boundary` section can contain three different constructs that define the boundary: `lines()`, `curve()` or `composite()`, described in the following sections.

3.1.1 The lines() Boundary Construct

In case of single lines or a number of connected lines, a special construct `lines()` is used, for efficiency reasons. A line segment is defined as follows:

```
domain Line2D "A line segment"
  extends Cartesian2D;
  parameter Real x0=0, y0=0, x1=1, y1=1;
boundary
  lines({{x0,y0},{x1,y1}});
end Line2D;
```

The `lines()` construct contains an expression which is an array of points, defining the starting point, the intermediate points and the end point of the connected lines (see Figure 1).

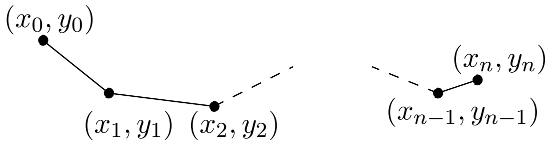

Figure 1: Connected lines that is described by the construct lines({{x0,y0},{x1,y1},...,{xn,yn}})

3.1.2 The curve() Boundary Construct

There are several alternative ways to specify the parametric expression that defines the boundary as a curve. Using the `where...in...` construct which already has been used to specify domains for expressions [14], the curve can be defined as follows:

```
domain Cartesian2D "For all 2D-domains"
  space Real x,y;
end Cartesian2D;

domain Circle2D "Circular with r=1"
  extends Cartesian2D;
boundary
  curve(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. The comma-separated list of expressions in the `curve()`-construct are used to calculate the Cartesian coordinates of the points on the curve. 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.

3.1.3 The composite() Boundary Construct

In many cases, the boundary of the problem domain is difficult to define as a single parametric curve, but is rather defined by a number of connected lines and curves. 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 `right`, `top`, `bottom`, and `left` (see Figure 2). These parts of the boundary can be instantiated in the declaration part of the domain class `Rectangle2D` as follows:

```
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);
boundary
  composite(right, top, left, bottom);
end Rectangle2D;
```

The domain `Rectangle2D` can be seen in Figure 2. 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 as domains, they represent different kinds of objects. The `Line2D` domain is not intended to be used as a domain by itself, but rather as a boundary component of another domain. 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.

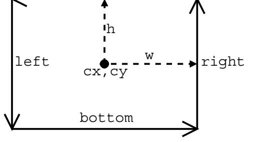

Figure 2: A rectangular domain `Rectangle2D`, defined using line segments. Note that the direction of the lines must be consistent.

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. A coefficient-based PDE base model can be defined as follows:

```
model PDE2D
  space Real x,y;
  Real u(x,y);
end PDE2D;

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

The variable u represents the unknown variable, a function of time and the space variables. All parameters can be constants or functions of the space variables. However, in this example, the coefficients da, c, a and f are restricted to be constants only, for clarity. The der operator is an operator in 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:

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

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

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

A Robin boundary condition, used in a heat problem to describe a boundary that is neither a perfect conductor nor a perfect insulator, can be written by first writing a general Robin boundary condition:

```
model Robin "Robin boundary condition"
  extends PDE2D;
  parameter Real c = 1;
  parameter Real q = 1;
  parameter Real g = 0;
equation
  nder((c*grad(u))) + q*u = g;
end Robin;
```

In mathematical notation, this equation is written as follows:

$$\frac{\partial}{\partial n}(c\nabla u) + qu = g$$

The operator nder() is a special operator that represents the derivative in the outward normal direction with respect to the associated domain boundary.

Other types of boundary conditions, e.g. Dirichlet and Neumann conditions, describing a perfect heat conductor and a perfect insulator, respectively, can be defined by extending the Robin class and setting the appropriate parameters to zero, as follows:

```
model Neumann
  extends Robin(q=0);
end Neumann;

model Dirichlet
  extends Robin(c=0);
end Dirichlet;
```

For heat transfer problems, a more specific version of the Robin boundary condition can be defined by inheriting the Robin class and adding application specific parameters and mapping them to the general parameters:

```
model HeatRobin "For heat transfer"
  extends Robin(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 HeatRobin;
```

The corresponding mathematical equation with these parameters is as follows:

$$\frac{\partial}{\partial n}(k\nabla u) = qh + hh(T_{inf} - u)$$

where qh is the source term, hh is the heat transfer coefficient and T_{inf} is the external temperature.

3.3 Problem definition

Once the models for the PDE and the boundary conditions are stated and the domain is defined, the problem can be put together by instantiating the PDE model, the boundary conditions, and the domain and associating the boundary conditions with the boundary parts. In order to associate boundary conditions and boundary elements, an implicit variable `bc` (short for boundary condition) is introduced in the restricted class `domain`. For each domain instance this variable is assigned the desired boundary condition. Similarly, a PDE is associated with a domain by instantiating the PDE model and assigning the instance to the variable `eq` (short for equation), also a builtin variable in the restricted class `domain`. The complete problem statement is then:

```
model PDEModel
  Neumann h_iso;
  Dirichlet h_heated(g=50);
  HeatRobin h_glass(hh=30000);
  HeatTransfer ht;
  Rectangle2D dom;
equation
  dom.eq = ht;
  dom.left.bc = h_glass;
  dom.right.bc = h_heated;
  dom.top.bc = h_iso;
  dom.bottom.bc = h_iso;
end PDEModel;
```

Here, a Dirichlet condition with a constant value of 50° for u is used to emulate a heat source on the right side of the domain, Robin condition is used for a non-isolating glass layer on the left side, and Neumann condition is used for the isolated top and bottom sides. The PDE model `HeatTransfer` is instantiated as `ht`, and used in the interior of the domain `dom`, which is an instance of the `Rectangle2D` class.

4 Results

The PDE extensions discussed in Section 3 were implemented in the prototype Modelica translator generated from a Natural Semantics specification of Modelica (see Section 4.2). A heat transfer example is solved in the following section in order to demonstrate the PDE extensions and the prototype. In this example, a stationary problem is solved, because the PDE solver currently used with the prototype does not handle time-dependent problems.

4.1 Example

A stationary heat conduction problem is considered. The problem is described by Poisson's equation:

$$-\nabla \cdot (c\nabla u) = g$$

where c is the heat conductivity coefficient, and g is the source term. In this example, c is set to 1 and g is set to 0.

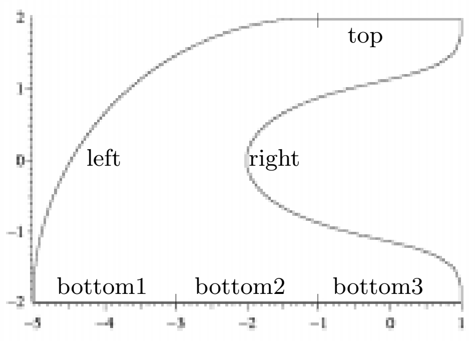

Figure 3: The problem domain with its different boundary sections.

The equation is solved on the domain shown in Figure 3. The left side of the domain is defined by a 90° arc, using an instance of a more general version of the domain class `Circle2D` defined in Section 3.1.2. The right side is defined by a Beziér curve with six control points, as the instance `right` of type `Bezier2D` defined below. The PDE-Modelica code for defining Beziér curves using De Casteljau's Algorithm follows:

```
function bezier
  constant Integer n=6;
  input Real px[n];
  input Real u;
  output Real res;
  Real qx[n];
algorithm
  for i in 1:n loop
    qx[i] := px[i];
  end for;
  for k in 1:n-1 loop
    for i in 1:(n-k) loop
      qx[i] := (1-u)*qx[i] + u*qx[i+1];
    end for;
  end for;
  res := qx[1];
end bezier;

domain Bezier2D
  constant Integer n=6;
  parameter Real px[n];
  parameter Real py[n];
  space Real u;
```

```
    BoundaryCondition bc;
  boundary
    curve(bezier(px,u), bezier(py,u));
  end Bezier2D;
```

The complete description of the domain for the heat transfer example in PDE-Modelica is:

```
  domain HeatExampleDomain
    extends Domain2D;
    Circle2D left(x0=-1, y0=-2, ra=4,
                  a=PI/2, b=PI/2);
    Bezier2D right(px={1.0, 1.3, -4.0,
                       -4.0, 1.3, 1.0},
                   py={-2.0, 0.0, -3.0,
                        3.0, 0.0, 2.0});
    Line2D top(x0=1, y0=2, x1=-1, y1=2);
    Line2D bottom1(x0=-5, y0=-2,
                   x1=-3, y1=-2);
    Line2D bottom2(x0=-3, y0=-2,
                   x1=-1, y1=-2);
    Line2D bottom3(x0=-1, y0=-2,
                   x1=1, y1=-2);
  boundary
    composite(right, top, left,
              bottom1, bottom2, bottom3);
  end HeatExampledomain;
```

At the right border, the Robin boundary condition is used, in order to model heat flow through the boundary that is proportional to the temperature difference. The temperature outside the domain is set to 20°. The middle part of the bottom border is used as a heat source, with a Dirichlet boundary condition $u = 50$. The other parts of the bottom border as well as the left and the top borders are perfectly insulated, using the homogeneous Neumann boundary conditions.

A plot of the solution can be seen in Figure 4. This example was solved using the finite element solver `rheolef` [15], and `bamg` [1] was used as the mesh generator.

4.2 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 [16] for solving the PDEs. The different modules of this environment can be seen in Figure 5. Here, the models are written in a Mathematica style Modelica 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 calls the solver generator that generates a finite element solver in C++. 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.

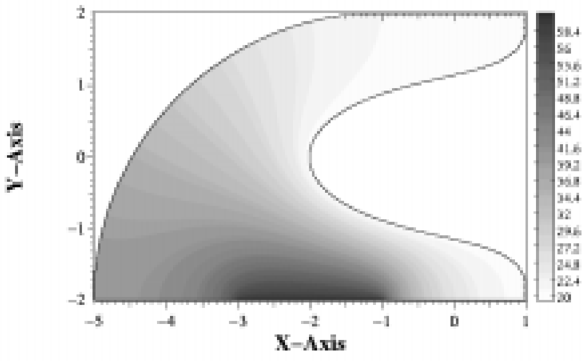

Figure 4: A stationary heat transfer example. The middle section of the bottom border is a heat source with $u = 50°$, the curved right border is non-insulated with outside temperature $20°$, and the other borders are insulated.

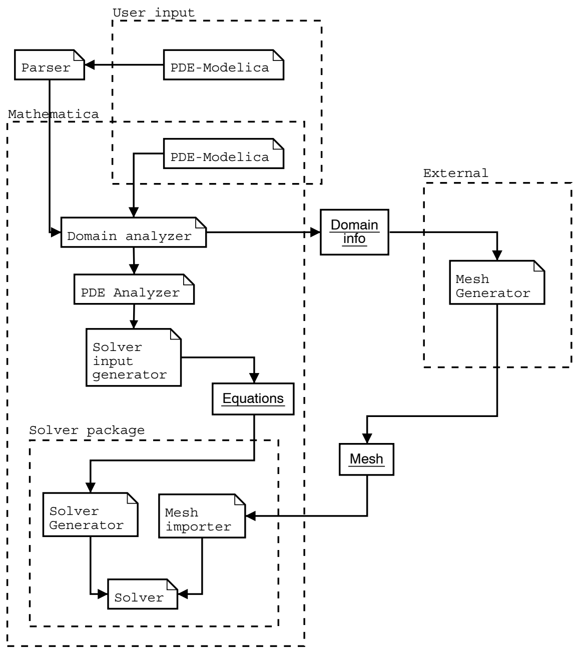

Figure 5: The PDE-Modelica prototype in the MathModelica environment

The other prototype environment consists of a Modelica parser, a compiler generated by the RML [13] system from a Natural Semantics description of Modelica, an external mesh generator and a PDE solver. The structure of this environment can be seen in Figure 6. The compiler generates C++ code from the PDE-Modelica description. The resulting code generates the discretized boundary at runtime, calls the mesh generator to triangulate the domain and finally calls the finite element solver.

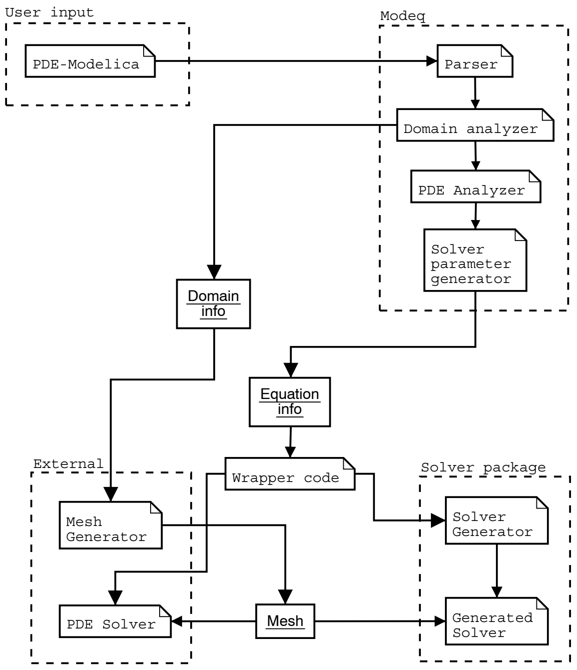

Figure 6: The PDE-Modelica prototype using the Modelica translator generated from Natural Semantics specification of Modelica in RML.

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.

5 Conclusions and Future Work

We have presented a design for specifying PDE problem domains in Modelica by expressing the boundary of the domain using lines and parametric curves. We have also shown a simple example of hierarchically defined PDE model and boundary conditions and how these can be used in a problem specification together with a domain definition.

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 different kinds of models needs to be considered. Support for combination of domains using set operations such as union, intersection, etc., and composition of domains into bigger domains using connect statements with different PDE models on each partial domain is another possible future extension.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] P. Fritzson, J. Gunnarsson, and M. Jirstrand. MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming. In *Proc.*

of the 2nd International Modelica Conference, Munich, March 2002.

[9] 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.

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

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

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

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

[14] 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.

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

[16] K. Sheshadri and P. Fritzson. A General Symbolic PDE-Solver Generator: Explicit Schemes. *Accepted for publication in Scientific Programming*, 2001.

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