PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations

by

Levon Saldamli

Submitted to the School of Engineering at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2002
PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations

by
Levon Saldamli

Dec. 2002
ISBN 91-7373-560-4
Linköping Studies in Science and Technology
Thesis No. 990
ISSN 0280-7971
LiU-Tek-Lic-2002:63

ABSTRACT

This thesis describes initial language extensions to the Modelica language to define a more general language called PDEModelica, with built-in support for modeling with partial differential equations (PDEs). Modelica® is a standardized modeling language for object-oriented, equation-based modeling. It also supports component-based modeling where existing components with modified parameters can be combined into new models. The aim of the language presented in this thesis is to maintain the advantages of Modelica and also add partial differential equation support.

Partial differential equations can be defined using a coefficient-based approach, where a predefined PDE is modified by changing its coefficient values. Language operators to directly express PDEs in the language are also discussed. Furthermore, domain geometry description is handled and language extensions to describe geometries are presented. Boundary conditions, required for a complete PDE problem definition, are also handled.

A prototype implementation is described as well. The prototype includes a translator written in the relational meta-language, RML, and interfaces to external software such as mesh generators and PDE solvers, which are needed to solve PDE problems. Finally, a few examples modeled with PDEModelica and solved using the prototype are presented.

This work has been supported by the Swedish Foundation for Strategic Research in the EC-SEL graduate school and the VISIMOD project, the European Commission in the RealSim project, and by Vinnova in the VISP project.
Acknowledgments

First of all I would like to thank my supervisor Peter Fritzson for the inspiration for doing the work and writing this thesis, and for improving my writing. I would also like to thank Vadim Engelson, Bernhard Bachmann and Mikael Adlers for their guidance during this work.

Many thanks go to my colleagues at PELAB (Programming Environments Laboratory) for contributing to an interesting and stimulating environment with lively and sometimes too long coffee break discussions. In particular, I would like to thank the members of the “pelab-modelica” group, and especially Peter Aronsson and Peter Bunus for many meetings and exchange of ideas. Many thanks also to Andrzej Bednarski and Jon Edvardsson for \LaTeX{} support. Special thanks to Bodil Mattsson Kihlström for her help with all the administrative issues. Also thanks to Lillemor Wallgren and Bodil Carlsson at IDA for their support with other administrative tasks, and to Ivan Rankin at IDA for language review of this thesis.

I would also like to thank the members of the Modelica Association, for developing the Modelica language, and especially Hans-Jürg Wiesmann and Hubertus Tummescheit for their ideas and comments on this work.
# Contents

1. Introduction 1  
   1.1. PDE-based Model Example 2  
       1.1.1. Connection to ODE/DAE models 3  
   1.2. Contributions 5  
   1.3. Overview of the Thesis 6  

2. Background 7  
   2.1. Modelica 7  
   2.2. Partial Differential Equations 14  
       2.2.1. Classification 16  
   2.3. Solution of Partial Differential Equations 17  
       2.3.1. Finite Difference Methods 17  
       2.3.2. Finite Element Methods 18  
       2.3.3. Finite Volume Methods 19  
       2.3.4. Method of Lines 20  

3. Related Work 21  
   3.1. Libraries and Programming Language-Based Packages 21  
       3.1.1. Diffpack 22  
       3.1.2. Overture 22  
       3.1.3. Compose 22  
   3.2. High Level Packages and Problem Solving Environments 23  
       3.2.1. gPROMS 23  
       3.2.2. PELLPACK 24  
       3.2.3. PDESpec 25  
       3.2.4. FEMLAB 26  
   3.3. Discussion 28  

4. PDEModelica 31  
   4.1. Problem Statement 31  
   4.2. PDE Problem Specification 31  
   4.3. Domain Geometry Definition 32
Contents

C. Problem Formulation Generated for FEMLAB 89
D. PDE Formulation Generated for Mathematica Based Solver 91
Chapter 1.

Introduction

In mathematical modeling of a physical system, the behaviour of different parts of the system is typically modeled using differential equations or a system of differential and algebraic equations, i.e. algebraic equations containing derivatives.

For large scale models, some properties of the physical system that is modeled are usually approximated; for example when controlling the temperature of a fluid in a container, the temperature can be assumed to be the same everywhere inside the container. This way, the space dependency is eliminated, resulting in models containing only ordinary differential equations (ODEs), or differential and algebraic equations (DAEs), i.e. equations where all derivatives are with respect to time. This greatly simplifies the solution of the equations and the simulation.

However, sometimes a more detailed model with explicit space dependency is needed for the whole system or some part of the system. Such space-dependent models are used to study the behaviour on a smaller scale, and contain partial derivatives, i.e. derivatives with respect to space variables like \( x \), \( y \) and \( z \) in a Cartesian coordinate system. Differential equations containing partial derivatives are called partial differential equations (PDEs), and are used in many fields to model physical behaviour, for example structural mechanics, computational fluid dynamics and electrostatics.

There are many tools for modeling and simulation with ordinary differential equations as well as with partial differential equations. However, most of these tools are specialized for certain application domains or special kinds of models.

An effort to define a general, application domain and tool-independent modeling language is made by Modelica Association, with the Modelica language as a result. Modelica® [15, 27] is a standard language designed for component-based modeling, where previously defined models can be reused as components in new models. Object oriented constructs, equation-based and declarative modeling support, as well as a connection concept make
Chapter 1. Introduction

this possible. Currently, the Modelica language only supports models with ordinary differential equations. Several Modelica implementations and simulation tools exist [2,5,16,18].

Thanks to the progress in computer performance, the use of detailed models containing partial differential equations is becoming increasingly common. The aim of this work is to define a language based on Modelica, in order to handle partial differential equation based models in addition to differential and algebraic equation based models. This language is called PDE-Modelica, and this thesis discusses the initial language extensions needed for the desired support.

1.1. PDE-based Model Example

Consider a simplified model of heat distribution in a rectangular room where only the distribution in the x and y directions is studied and the temperature in the z direction is assumed to be constant. A heater is installed on one of the walls and there is a window on another wall. The domain of this problem can be seen in Figure 1.1. A heater is represented by assigning constant temperature to the middle part of the upper edge. The window is modeled by a non-zero heat flow through the middle left edge that is proportional to the temperature difference between both sides of the edge. The rest of the edges are modeled as insulated edges, i.e. no heat flow occurs through these edges.

The stationary heat conduction problem can be modeled using the Laplace equation if no heat sources exist in the room. The boundary conditions are the following:

\[ T_{out} = 20^\circ C \]

\[ T = 30^\circ C \]

Window

Heater

Figure 1.1.: Example of stationary heat conduction.

The stationary heat conduction problem can be modeled using the Laplace equation if no heat sources exist in the room. The boundary conditions are the following:
1.1. PDE-based Model Example

- a Dirichlet\(^1\) boundary condition for the heated wall,
- a Neumann\(^2\) boundary condition for the insulated walls, and
- a Robin\(^3\) (mixed) boundary condition for the window which is poorly insulated.

This problem can be modeled in PDEModelica with the geometry and the model description shown in Figure 1.2. The domain geometry is defined by describing its boundary in a specific direction in order to find out on what side of the boundary the actual domain is. The boundary is built up of several sections defined as named components in the domain description. Hence, when defining the PDE model, boundary conditions can be assigned to each section using the name of each boundary component. The definition of the boundary conditions and the PDE are not shown in this example; they can be defined as described in Chapter 4. The solution to a similar example can be found in Section 6.3.

1.1.1. Connection to ODE/DAE models

The previous example shows a stationary model of heat conduction. Another application is to study the heat distribution over time, starting from an initial state. A time derivative is then added to the Laplace equation. Furthermore, using a time-dependent model, a system with active temperature control can be modeled. A temperature sensor can be approximated by reading the computed temperature at some point in the domain, and the temperature can be used as a signal to a controller that controls a heater. The overview of the system is illustrated in Figure 1.3.

The controller is modeled by an ODE, resulting in a problem consisting of a PDE part modeling the temperature distribution, and an ODE part modeling the controller. This can be compared to a simplified system where the temperature is assumed to be the same at all points. The temperature distribution can then be modeled using an ODE based model, which results in a complete system with only ODEs. Coupled ODE and PDE models are needed in cases where some part of an ODE model is replaced with a PDE model because the spatial variations are important for the model.

This example is simplified by the fact that the sensor is replaced with a simple result reading. In other situations, the connection between the PDE

---

\(^1\)Value of the unknown variable known on the boundary. See also Section 2.2.
\(^2\)Value of the outward normal derivative of the unknown variable is known on the boundary. The outward normal derivative is the space derivative in the outward normal direction. See also Section 2.2.
\(^3\)The sum of the unknown variable and its outward normal derivative is known on the boundary. See also Section 2.2.
Chapter 1. Introduction

```modelica
domain Rectangular "Geometry"
Line2D right (x0= 3, y0=-2, x1= 3, y1= 2);
Line2D top1 (x0=right.x1, y0=right.y1, x1= 1, y1= 2);
Line2D top2 (x0=top1.x1, y0=top1.y1, x1=1, y1= 2);
Line2D top3 (x0=top2.x1, y0=top2.y1, x1=-3, y1= 2);
Line2D left1 (x0=top3.x1, y0=top3.y1, x1=-3, y1= 1);
Line2D left2 (x0=left1.x1, y0=left1.y1, x1=-3, y1=-1);
Line2D left3 (x0=left2.x1, y0=left2.y1, x1=-3, y1=-2);
Line2D bottom(x0=left3.x1, y0=left3.y1, x1= 3, y1=-2);
boundary
  composite(right, top1, top2, top3, left1, left2, left3, bottom);
end Rectangular;

model PDEModel "Equations and boundary conditions"
  HeatRobin heatloss(Tout=20);
  Dirichlet constheat(c=30);
  Neumann isolated;
  HeatConduction ht;
  Rectangular dom;
  equation
    dom.eq = ht;
    dom.left1.bc = isolated;
    dom.left2.bc = bc_robin;
    dom.left3.bc = isolated;
    dom.right.bc = isolated;
    dom.top1.bc = isolated;
    dom.top2.bc = constheat;
    dom.top3.bc = isolated;
    dom.bottom.bc = isolated;
end PDEModel;
```

Figure 1.2.: The PDEModelica code to define the heat conduction problem in Section 1.1. Predefined PDE and boundary condition models have been left out. The syntax is explained in Chapter 4.
1.2. Contributions

The result of this work is preliminary design and implementation of an object-oriented modeling language, PDEModelica, with support for

- object-oriented modeling with PDEs and boundary conditions,
- complex geometry description with lines, polygons, parametric curves and a combination of these,
- component-based geometry definition,
- hierarchical modeling and decomposition with a general connection concept, and
- modeling of combined ODE/DAE and PDE problems.
Chapter 1. Introduction

We believe that the combination of these aspects in a single language is rather new.

A prototype translator and solver environment is set up as well, with a basic PDE solver interface for adding new solvers and mesh generators.

Some of the work discussed in this thesis has been published previously in the following publications:


1.3. Overview of the Thesis

The thesis is organized as follows. Chapter 2 contains background information relevant for the thesis: a short overview of the Modelica language, a basic introduction to partial differential equations and some existing numerical solution methods for partial differential equations.

Chapter 3 presents an overview of related work. Different low-level and high-level tools with varying modeling language support are summarized.

Chapter 4 describes the main work and goes through the extensions to the Modelica language for domain description and PDE definition.

Chapter 5 provides the details about the implementation of the prototype, which implements the extensions defined in Chapter 4.

Chapter 6 contains examples with the PDEModelica code and the solutions.

Finally, in Chapter 7 some conclusions and possible future work directions are discussed.
Chapter 2.

Background

The basic concepts of Modelica are presented in the first part of this chapter. In the second part, the topic of partial differential equations (PDEs) and classification of PDEs are briefly presented, and finally an overview of different numerical solution methods is given.

2.1. Modelica

Modelica [17, 27] is a modeling language for equation-based, object-oriented modeling and simulation of physical systems. Using object-oriented concepts, it allows hierarchical, component-based modeling which in turn makes reuse of existing models possible. The general modeling concepts in Modelica allow it to be used in different application domains and in multi-domain modeling, for example when defining a combined electrical and mechanical model. There is a free Modelica Standard Library [28] and other free Modelica libraries with packages of existing models from different domains that can be used as components in specific models. Non-causal modeling, i.e. modeling using equations - not assignment statements, allows single models to be used in different data flow contexts since variables are not explicitly declared as input or output but this information is rather derived from the context where the model is used. In his PhD thesis, H. Tummescheit [45] discusses design and implementation of modeling libraries using Modelica.

Models in Modelica are hierarchically built from submodels that are defined separately, which in turn can be built in the same way. Hence, a model can contain one or more instances of other models with different set of parameter values for each instance, and a set of connections between these components. Additionally, each model can have variables of built-in types, and equations that define the relationship between these variables and also between variables in submodels.
Chapter 2. Background

Classes

The basic structural unit in Modelica is class. A class can contain other classes, variable declarations, equations and algorithms. An example of a class can be seen in Figure 2.1. Other keywords can be used to denote classes, restricted classes that are special cases of classes with some restrictions. These are record, type, connector, model, block, package and function. Different restricted classes and restrictions that apply to them can be seen in Table 2.1.

```model MyModel "Short description of this model"
   Real x,y;
   equation
      x + y = 5;
      x + 2y = 11;
   end MyModel;
```

Figure 2.1.: Example of a Modelica class.

<table>
<thead>
<tr>
<th>Restricted class</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>No equations are allowed.</td>
</tr>
<tr>
<td>type</td>
<td>May only be used to extend primitive types.</td>
</tr>
<tr>
<td>connector</td>
<td>No equations are allowed.</td>
</tr>
<tr>
<td>model</td>
<td>Model instances may not be used in connections.</td>
</tr>
<tr>
<td>block</td>
<td>Fixed causality. Each variable must be declared as input or output.</td>
</tr>
<tr>
<td>package</td>
<td>May only contain classes and constants. Is allowed to import from.</td>
</tr>
<tr>
<td>function</td>
<td>Similar to block. No equations, at most one algorithm section. May be called using positional arguments. May be recursive.</td>
</tr>
</tbody>
</table>

Table 2.1.: Restricted classes in Modelica.

Variable Declarations

Variable declarations consist of a type and a variable name. The type can be one of the primitive types or another class name. The primitive types are Real, Integer, String, Boolean and Enumeration. Each declaration can also have type modifiers, such as the type variability prefix constant, discrete, or parameter, or a causality prefix, input or output.
Variables with the `constant` or `parameter` prefix keep their value constant during a simulation, i.e., they are constants. The difference between parameters and constants is that values of parameters can be changed before each simulation without needing to recompile the Modelica code. Variables of type `Real` declared without the constant or parameter modifiers are implicitly time-dependent, i.e., functions of time. Time derivatives of variables can be represented using the `der()` operator. Using the time derivatives, ordinary differential equations can be expressed, and full systems of differential and algebraic equations (DAEs) can be specified.

Subtyping and Inheritance

Modelica models can be defined using inheritance. The `extends` clause is used together with a class name to inherit that class. Multiple inheritance is allowed using several `extends` clauses, one for each inherited class. Inheritance is equivalent to inserting the contents of the inherited class at the place where the `extends` clause resides.

Type equivalence in Modelica is defined as follows: Two types T and U are equivalent if

- they denote the same primitive type (see previous section), or
- T and U are classes containing the same elements (according to their names) and the elements types are equivalent.

Subtypes in Modelica are defined independently from the inheritance mechanism. A class C is a subtype of a class S if

- S and C are type equivalent, or
- both of the following statements hold:
  - every public element of S also exists in C (according to its name).
  - the type of each such element in C is a subtype of the type of the corresponding element in S.

If a class C is a subtype of a class S then S is called the supertype of C. Subtypes and supertypes do not necessarily need to be in an inheritance hierarchy, but a class that is inherited from, a base class, is a supertype of a class that inherits from it, a derived class. An example is shown in Figure 2.2.

The `TempResistor` class cannot extend the `Resistor` class because it has a different equation in the equation section, but it is still a subtype of
Chapter 2. Background

partial model OnePort
  Pin p,n;
  Voltage v "Voltage drop";
equation
  v = p.v - n.v;
  p.i + n.i = 0;
end OnePort;

model Resistor;
  extends OnePort;
  parameter Real R(unit="Ohm") "Resistance";
equation
  v = p.i * R;
end Resistor;

model TempResistor "Temperature dependent resistor" extends OnePort;
  parameter Real R(unit="Ohm") "Resistance at reference temperature";
  parameter Real RT(unit="Ohm/degC") = 0 "Temperature dependent resistance";
  parameter Real Tref(unit="degC") = 20 "Reference temperature";
  Real Temp = 20 "Actual temperature";
equation
  v = p.i * (R + RT * (Temp - Tref));
end TempResistor;

Figure 2.2.: Subtyping in Modelica. TempResistor does not inherit from Resistor, but it is a subtype of Resistor. They both inherit from OnePort.
Resistor because it contains all the elements in Resistor, and additional elements. Both classes inherit from the OnePort\(^1\) class, though.

**Modifications**

Modifications in Modelica are used to modify parameter values when declaring instances of classes. Since models are built up hierarchically, modifications can be overridden, in which case the topmost (outermost) modification is applied. A modification can also be hierarchical, in order to modify a lower level parameter directly. Each modification consists of a component reference and an expression that is evaluated in the context where the declaration resides. Example of modifications can be seen in Figure 2.3. Declarations preceded by the `final` keyword are final elements and cannot be modified using modifications. The `final` keyword can also precede modifications, in which case it prevents further modifications of that variable in outer modifications.

```modelica
model ModelA
    parameter Real pa = 0; // default value
end ModelA;
model ModelB
    ModelA compa(pa=1); // modification
    parameter Real pb = 0;
end ModelB;
model MyModel
    parameter Real mypa=3, mypb=4;
    ModelB compb(pb=mypb, compa(pa=mypa)); // hierarchical modification
end MyModel;
```

Figure 2.3.: Modifications in Modelica. The resulting value of the variable `compb.compa.pa` is `mypa`, i.e. 3.

Modification can also be applied when extending another class, together with the `extends` clause. For example:

```modelica
model MyNewModel
    extends MyModel(mypb=5);
```

When instantiating components of `MyNewModel` the default value of `mypb` will be 5.

\(^1\)The term OnePort is used by specialists in the electrical modeling community to denote electrical components with two physical connection points.
Chapter 2. Background

Equations and Algorithms

The keywords equation and algorithm denote equation and algorithm sections, respectively. There can be several such sections of each kind in a class. During compilation, all equation sections are merged into a single equations section.

Equation sections can contain standard equation clauses, connect equations, conditional equations, for equations and when equations. Standard equations consist of two expressions and the equality operator denoted by =. The assignment operator := is not allowed in equation sections, whereas the equality operator = is not allowed in algorithm sections. Besides the common operators like arithmetic operators and function calls, Modelica has if-expressions, which can be used in situations like:

\[
equation
x = \text{if } (z > 0) \ 2*z \text{ else } (-2*z);
\]

Connect equations consist of the connect() operator with two arguments that are references to connectors. Connectors are instances of connector restricted classes and usually contain two kinds of variables, flow (e.g. current) and non-flow (e.g. potential) variables, the former being denoted by the flow keyword. Connect equations are translated into standard equations during compilation, where for each connection the non-flow variables of the involved connectors are set equal, and the sum of all the flow variables is set to be equal to zero. Figure 2.4 illustrates the use of connectors and connections, and the resulting equations.

Conditional equations are written using the if-elseif-else construct with equations in the body. for equations are useful for repetitive equation structures involving arrays. when clauses are used for discrete event simulation.

Algorithm section can contain assignments, conditional statements, for-loops, while-loops and when-statements. The difference from equation sections is that algorithm sections contain assignments instead of equations, and the contents of the algorithms are executed sequentially. An assignment consists of a variable reference, the assignment operator := and an expression.

Replaceable Elements

Elements in classes in Modelica can be declared as replaceable, in which case they can be replaced by another type of element that is type compatible when the containing class is extended or instantiated. Generic classes with type parameters is supported in Modelica through the replaceable mechanism.
2.1. Modelica

(a) Visual view of a connector and connection example.

connector Pin "Electrical pin"
  flow Current i;
  Voltage v;
end Pin;

model OnePin
  Pin p;
end OnePin;

model Test
  OnePin a, b, c;
equation
  connect(a.p, b.p);
  connect(b.p, c.p);
end Test;

(b) Textual view of a connector and connection example.

equation
  a.p.i + b.p.i + c.p.i = 0;
  a.p.v = b.p.v;
  b.p.v = c.p.v;

(c) Resulting equations

Figure 2.4.: Example of connectors and connections and the resulting simple equations after compilation.
Chapter 2. Background

The type restriction of a redeclaration is that the new declared type must be a subtype of the original declared type, a constraining type.

2.2. Partial Differential Equations

Differential equations are commonly used in mathematical models of physical phenomena that are distributed in time and/or space. Examples include heat transfer, fluid flow, structural mechanics, wave propagation, etc. Such mathematical models relate certain variables, dependent variables and their derivatives, to other variables, independent variables.

In some cases a variable that is distributed in space and/or time, e.g. a temperature field, can be approximated by a scalar variable. For example, when the temperature of a fluid in a container is studied, the temperature could be assumed to be the same throughout the entire container at each specific time instant, and one can study the temperature change as a function of time based on the incoming and outgoing fluid flow. Since the simplified model depends only on one variable, time, derivatives that occur are time derivatives, and the equations of the model are ordinary differential equations (ODEs).

In more complex models when the temperature distribution over the container is studied with a different temperature at different positions inside the container, the temperature is a function of the independent variables representing the space coordinates inside the container, and also of the time variable if time-dependent behaviour is studied. Derivatives that occur in such models are partial derivatives with respect to one of the variables, the space coordinates or time.

Differential equations involving partial derivatives are thus called partial differential equations (PDEs). If time dependency is present, the models are called time-dependent problems, otherwise stationary.

The order of a partial differential equation is defined to be the highest differentiation order that occurs in that equation. The most commonly used PDEs in models of many practical systems are second-order PDEs, containing second order derivatives. A partial derivative can be stated in different ways. The first-order partial derivative of a dependent variable $u$ with respect to the independent variable $x$ is represented using the following three variants of mathematical notation:

$$\frac{\partial u}{\partial x} = \partial_x u = u_x$$

A second order derivative can be written accordingly as

$$\frac{\partial^2 u}{\partial x^2} = \partial_{xx} u = u_{xx}$$
2.2. Partial Differential Equations

The second differentiation can be done with respect to a different variable \( y \) than the first differentiation, e.g. \( \partial_{xy}u = u_{xy} \), which is also a second order derivative, a mixed partial derivative.

A general, second order PDE with an unknown dependent variable \( u \) and the independent variables \( x \) and \( y \) can be stated as

\[
f(u, u_x, u_{xx}, u_y, u_{yy}, u_{xy}, u_{yx}, x, y) = 0 \quad x \in \Omega \quad (2.1)
\]

The definition domain, i.e. the geometric region on which the PDE is defined, denoted \( \Omega \), is in most cases a closed region in space as defined by the independent variables, see Figure 2.5.

![Figure 2.5: The definition domain \( \Omega \), its boundary \( \partial \Omega \) and the outward normal vector \( \pi \).](image)

Boundary conditions in a PDE problem specify the behaviour of the model on the boundary of the domain \( \Omega \), denoted \( \partial \Omega \), and usually contain derivatives of one order less than the PDE. A general boundary condition corresponding to equation (2.1) can be expressed as

\[
g(u, u_x, x) = 0 \quad x \in \partial \Omega \quad (2.2)
\]

Boundary conditions usually contain a directional derivative instead of the partial derivative with respect to one variable. A common directional derivative is the outward normal derivative, which is obtained by the scalar product of the outward normal on the domain boundary and a vector of the partial derivatives. The outward normal derivative is denoted \( \partial_n \) and is commonly used in boundary conditions to specify e.g. heat flux through the boundary.

Three kinds of boundary conditions that often occur have been given special names:

- **Dirichlet**: \( u = g \) on \( \partial \Omega \)

- **Neumann**: \( \frac{\partial}{\partial n}u = g \) on \( \partial \Omega \)

- **Robin**: \( \frac{\partial}{\partial n}u + qu = g \) on \( \partial \Omega \)

15
Chapter 2. Background

In cases where the right-hand side is zero, the corresponding condition is called homogeneous, otherwise non-homogeneous.

2.2.1. Classification

PDEs are classified based on linearity properties. An equation expressed using the linear operator\(^2\) \(\mathcal{L}(\cdot)\) in the form

\[
\mathcal{L}(\alpha u + \beta v) = 0
\]

is linear if

\[
\mathcal{L}(\alpha u + \beta v) = \alpha \mathcal{L}(u) + \beta \mathcal{L}(v)
\]

If a PDE with the dependent variable \(u\) and the independent variables \(x\) and \(y\) can be expressed in the following form:

\[
a u_{xx} + 2b u_{xy} + c u_{yy} = d
\]

(2.3)

where the coefficients \(a\), \(b\) and \(c\) are constants, the PDE is linear. If the coefficients are functions of the independent variables only, \(x\) and \(y\) in this example, the PDE is semi-linear, and if the coefficients also depend on \(u\) or its first-order derivatives the PDE is quasi-linear. Equations that cannot be expressed linearly in the second order derivatives as in equation (2.1) are nonlinear. In this text, semi-linear PDEs will be regarded as linear, while quasi-linear PDEs will be classified together with the nonlinear.

If \(a > 0\), Equation (2.3) is classified further into three categories depending on the value of \(b^2 - ac\):

- **Elliptic**: \(b^2 - ac < 0\)
- **Parabolic**: \(b^2 - ac = 0\)
- **Hyperbolic**: \(b^2 - ac > 0\)

Elliptic equations typically occur in stationary heat conduction models, whereas parabolic equations arise in time-dependent models. The wave equation used for modeling propagation of waves such as sound waves in gas or electro-magnetic waves is an example of a hyperbolic equation.

\(^2\)An example of a linear operator is the derivative operator.
2.3. Solution of Partial Differential Equations

A very small number of PDE problems can be solved symbolically, and even fewer of these are solvable with practically useful boundary conditions. For this reason, much research has been done on numerical solution of PDEs and different methods have been developed. Unlike for solution of ordinary differential equations, there is no unified method for solution of general PDE problems, however, and an appropriate numerical solver must be selected for the specific kind PDE being solved.

The solution method depends on whether a stationary or a time-dependent PDE is solved. Only space discretization is needed for stationary problems, whereas time-dependent problems need both space and time discretization. The methods described below, except Section 2.3.4, are used for space discretization. Time discretization can be done using for example finite differences, or the PDE can be discretized in space first and the resulting ODEs can be solved with traditional ODE solvers, as described in Section 2.3.4.

2.3.1. Finite Difference Methods

In order to find the unknown function, the domain, i.e. the geometry on which the problem is defined is discretized into a set of grid points and the values of the function at these points are calculated. When the domain is discretized, the partial derivatives can be approximated by equations using the values at the grid points. Using Taylor’s theorem, different approximations can be derived [43]. The first-order derivative of \( u \) with respect to \( x \) using a grid with the distance \( h \) between the points can be approximated by for instance:

\[
\frac{\partial u}{\partial x} = \frac{u(x + h) - u(x - h)}{2h} + O(h^2)
\]

Here, \( O(h^2) \) represents the approximation error. This equation is called a central-difference approximation. Other approximations can be derived, for example

\[
\frac{\partial u}{\partial x} \approx \frac{u(x + h) - u(x)}{h}
\]

which is called a forward-difference formula, and

\[
\frac{\partial u}{\partial x} \approx \frac{u(x) - u(x - h)}{h}
\]

which is called a backward-difference formula. Higher-order derivatives can be approximated similarly. The derivatives in the PDE are then replaced by the approximations and repeated for each grid point, generating an equation system with the values of \( u \) at the grid points as unknowns. The boundary
Chapter 2. Background

conditions are applied at the grid points neighboring the boundary, depending on the kind of boundary conditions. With Dirichlet boundary conditions, the values on the boundary are known and can be used directly. With Neumann and Robin conditions, the derivative is approximated in the same way as in the PDE and equations involving the grid points outside the domain are generated, that can be used to express the unknown values on the boundary.

When solving time-dependent problems, the time derivatives can be approximated with finite differences as well. Note however, that when using finite difference methods, the discretization steps must be chosen according to certain conditions in order to maintain stability.

2.3.2. Finite Element Methods

A common way of solving a PDE numerically is to approximate the dependent variable $U$ with a series of known basis functions, also known as trial functions, and then to find out the coefficients that satisfy the equation as well as possible, by minimizing the error introduced by the approximation. If the basis functions are denoted $v_k$, the approximation $\hat{u}$ of $u$ can be stated as:

$$\hat{u}(x) = \sum_{k=0}^{N} a_k v_k(x)$$  \hfill (2.4)

Different solution methods use different norms of how close the approximation is to the unknown function, when calculating the coefficients. Since there are a finite number of basis functions and finite number of unknown coefficients, it is possible to numerically calculate the coefficients, given the PDE and known values at some points, i.e. the boundary conditions.

The Galerkin method is a basic method for calculating the unknown coefficients. For a partial differential equation of the form

$$L(u(x)) = 0$$

where $L()$ is a linear operator containing $u$ and its derivatives, a residual error $R$ of an approximation $\hat{u}$ is defined by

$$R(\hat{u}(x)) = L(\hat{u}(x))$$

The residual error measures how well the approximation satisfies the partial differential equation. If it is identical to zero, i.e. $R(\hat{u}(x)) \equiv 0$ for all $x$ in the domain, then the approximation is the exact solution. A criterion for measuring how close a function is to being zero is to look at its orthogonality to a chosen set of functions. Orthogonality of two functions $f(x)$ and $g(x)$ is defined as

$$\int_{\Omega} f(x) g(x) \, dx = 0$$
2.3. Solution of Partial Differential Equations

The residual of the exact solution \( R(u(x)) \) is orthogonal to all functions according to this definition. The residual of the approximation is checked against a set of test functions from an appropriate test space. It is sufficient to check the basis vectors of the test space to assure orthogonality to all functions in the test space.

The Galerkin method uses the trial space as the test space as well, and the basis functions \( v_k \) are used as the test functions:

\[
\int_{\Omega} R(\hat{u}(x))v_k(x) \, dx \quad k = 0..N
\]

Substituting \( \hat{u} \) with its definition in (2.4) and calculating the integrals gives an equation system that can be solved to finally obtain the coefficients to the trial functions and the approximation to the solution.

The selection of the trial and test functions lead to different solution methods. Using polynomials as basis functions for the trial space requires global support from the polynomials, because each basis function contributes to the entire domain. In order to increase the accuracy of the approximation the degree of the polynomials must be increased.

A more flexible method is to divide the domain into \( m \) subdomains and use basis functions with local support, where each function \( v_k \) is zero outside of one subdomain and its neighboring domains. This method is called the finite element method, elements being the subdomains, and can be applied to the different solution methods using different test functions. Usually lower order polynomials can be used because local support simplifies the approximation. Instead of polynomials of higher degree, smaller elements can be used for better approximation, especially for the parts of the domain where the solution has steep changes. Subdivision of the domain can be performed in different ways, triangulation being a common method. Adaptive triangulation can also be used, to refine the mesh iteratively on parts of the solution where the error is too large.

2.3.3. Finite Volume Methods

As with the finite element method the domain is discretized with a mesh generator. Then, a control volume is defined over each grid point, such that the control volumes do not overlap, and the PDE is integrated over each control volume. This integral can be approximated by finite differences of adjacent grid points. The result is a discretization equation with the values of the dependent variable on the grid points.
2.3.4. Method of Lines

The method of lines is a solution method that converts a PDE into a set of ODEs involving only time-dependent functions and time-derivatives. The conversion is done by discretizing the PDE in space, leaving a number of unknowns and their time derivatives. For the space discretization, any of the methods described earlier can be used. For example, if the finite difference method is used, the space discretization leads to one unknown and its time derivative at each grid point on the domain, i.e. a set of ODEs.

One advantage of the method of lines is that advanced numerical solution methods exist for solving general ODEs that do not yet exist for PDEs. There are, for instance, solvers with automatic step adjustment to find a solution with required accuracy. Another advantage is that coupled systems containing both ODE and PDE based models become easier to solve because the space discretization of the PDEs results in ODEs that can be solved together with the already existing ODEs.

One disadvantage, though, is that the space discretization is independent of the error controlled step adjustment that is done in the ODE solution process. Thus, even though the ODE solver solves the given ODEs with a desired accurately, error introduced during the space discretization can be much larger if space discretization is performed without caution.

Also, the relation between the step sizes in the space and time domain must fulfill certain criteria in order to get a numerically stable solution. However, this is not checked by a pure ODE solver.
Chapter 3.
Related Work

This chapter contains an overview of work done on tools for modeling and simulation of partial differential equation based models, categorized by the user interaction level. The first category of tools requires the user to write the problem formulation and the solution algorithm in an ordinary programming language such as Fortran, C or C++, with library support from the tool. The second category provides a high-level interface, a problem description language or a graphical user interface, where the problem can be specified.

3.1. Libraries and Programming Language-Based Packages

There exist several libraries of PDE solver algorithms with solver routines suitable for different PDE problems. When using these libraries the PDE problem is formulated by defining the appropriate functions and data structures and solved by calling the appropriate solver routine. Programming experience and knowledge of programming languages is required to use such libraries. Mathematical and numerical knowledge is also needed to use appropriate solution methods. Repositories of solution algorithms exist, for example Netlib [29], with many stand-alone subroutines that are freely available for download. There are also commercial library packages, such as Diffpack [12], specialized for solving PDE problems with support for both finite difference and finite element methods.

Some PDE solver packages are written as frameworks, usually in object-oriented languages, with a level of abstraction that is higher than just using solver libraries directly. A specific PDE problem is solved by a program written in a programming language, usually C++ for object-oriented packages. Classes and objects from the framework are used directly in the program, or new extension classes are added to the framework and subsequently used.
Chapter 3. Related Work

3.1.1. Diffpack

Diffpack [12] is a C++ library package with data structures and numerical methods that are used to solve PDE problems. The package is divided into a kernel and application-specific toolboxes for use in different engineering areas. Diffpack includes support for solving sparse linear systems using iterative methods and solving non-linear systems, using finite element and finite difference methods with a collection of finite elements and finite difference schemes, adaptive meshes, domain decomposition methods and multi-grid methods. The package also includes support for administrative tasks, such as report generation, simulation result handling and data entry using a graphical user interface.

3.1.2. Overture

Overture [32] is a framework for writing PDE solvers in C++ using finite difference or finite volume methods. Built-in classes are used to represent domains, discretized domains, differentiation operators, solution vectors, and other data needed in a PDE solver system. The framework also includes grid generators, solvers, and graphical interface classes for plotting the results.

Overture uses overlapping grids to support composition of domain geometries. A grid generator is used to generate overlapping grids which are used for domains composed of several domain components. In a composite domain, common grid points are generated for the overlapping parts of the domain components that comprise the final domain. This method is also useful for moving overlapping grids where the domain components move over time, in which case the grid is regenerated each time a domain component is moved.

3.1.3. Compose

Compose [44] is an object-oriented framework built on top of Overture. The Compose framework has been developed using object-oriented analysis, design and implementation, with the objective of separating the classes used for problem formulation from the classes concerned with the numerical solution process. Hence, the equation classes, domain classes, and other classes that represent the mathematical model are separated from classes that handle discretization and solution of the PDEs. Thus, compared to Overture, where the formulation of the mathematical model and the numerical solution method are more tightly coupled, Compose has a higher level of abstraction.

However, there is no automatic solver selection support in Compose compared to many problem solving environments. Equations and equation dis-
cretizers must be associated for an equation to be discretized appropriately. The framework is extensible, so that new types of equations and corresponding equation discretizers can be added to the system.

3.2. High Level Packages and Problem Solving Environments

One of the first attempts to define a language for PDE problem formulation was PDEL [10]. In modern high-level language based or graphical user interface based PDE solving systems, the PDE and boundary conditions are specified directly, either in text form in a special purpose language or interactively in a specialized editor. The system automatically solves the PDE problem using a general solver or by selecting an appropriate solver from a set of existing solvers. The details about the solution process can be hidden from the user, although it is still possible to manually direct a solver or to select a solver to use.

3.2.1. gPROMS

\textit{gPROMS} is a general process modeling system for dynamic simulation of chemical processes, with support for combined lumped and distributed models [30]. \textit{gPROMS} uses a high-level language for modeling and simulation of mixed systems of integral, partial, and ordinary differential, and algebraic equations (IPDAEs) over rectangular domains.

\begin{verbatim}
MODEL TubularReactor

PARAMETER
    NbrComp AS INTEGER
    ...

    ReactorLength, ReactorRadius AS REAL

DISTRIBUTION_DOMAIN

    Axial AS ( 0 : ReactorLength )

    Radial AS ( 0 : ReactorRadius )

\end{verbatim}

Figure 3.1.: Example of a model definition in gPROMS.

A model definition in the gPROMS language is depicted in Figure 3.1. \textit{Axial} and \textit{Radial} in the model are the independent variables.

A dependent variable that is to be solved over this domain is declared as follows:
Chapter 3. Related Work

VARIABLE
Temp  AS DISTRIBUTION (Axial, Radial) OF Temperature

Here, Temp is of type Temperature that is defined elsewhere, and its geometric domain is the three-dimensional area defined by the independent variables Axial and Radial and their limits as defined in Figure 3.1. This declaration is similar to declaration of arrays in the gPROMS language, which is done using the array keyword:

F  AS ARRAY ( NbrComp ) OF Flowrate

For partial differentiation the partial operator is used. partial(expr, var) is the partial derivative of the expression expr with respect to the variable var (i.e. \( \partial_r Temp(z, R) \)), i.e. the partial operator supports differentiation of entire expressions, not only single variables.

For loops are used for restricting the applicability of an expression or equation to a part of the domain. An example can be seen in Figure 3.2, where a boundary condition is defined for the wall of the tubular reactor, which has an axial distribution but not radial. The boundary condition applies for values of \( z \) between 0 and ReactorLength.

FOR z:= 0 TO ReactorLength DO
- Kr * PARTIAL( Temp(z, ReactorRadius), Radial) =
  Uh * ( Temp(z, ReactorRadius) - TWall(z) )
END

Figure 3.2: For loops for restricting an equation to part of a domain as specified in gPROMS.

3.2.2. PELLPACK

PELLPACK [19] is a problem solving environment for specification, solution and post-processing of PDE problems. A PDE language is used to specify the PDE, the domain geometry, the boundary conditions, the discretization method, the solution method, and other method-specific parameters that are needed to solve the problem. There are also tools providing graphical user interfaces for specifying different parts of the problem, from which the specification in the same PDE language is generated.

The PDE language used in PELLPACK is an extension of the ELLPACK language. The ELLPACK language is divided into sections where the PDE problem is specified and the solution method is selected. The equation section contains the PDE written in mathematical form. The boundary section defines the geometry by defining the domain boundary. The boundary conditions are also written in the boundary section together with the
3.2. High Level Packages and Problem Solving Environments

domain boundary descriptions. Other sections are grid, discretization, solution and output which specify the modules to use in different stages of the solution process.

The ELLPACK language also supports embedded Fortran code in fortran sections. The PELLPACK language additionally has the mesh section to support the finite-element method, and sections for domain decomposition to support parallel solutions. The language is also extended to contain information produced from the graphical user interface that needs to be preserved.

In the ELLPACK language, there are predefined names for the solution variable and its derivatives, such as U, UX (∂x U), UYY (∂yy U) and the spatial variables X, Y and Z. These variables are used to specify the PDE. For example:

\[
\text{EQUATION. } U_{XX} + U_{YY} + 3.0*U_{X} - 4.0*U = \exp(X+Y)\sin(\pi*X)
\]

The boundary conditions can be specified in different ways, using lines and parametric curves. A circular domain is defined as in Figure 3.3, where the boundary section specifies the condition that the function is zero on a circle with radius 1 and center (1, 1). The boundary statement also defines the geometry of the problem. The equation and boundary parts of an ELLPACK program are declarative, whereas the rest grid, discretization, fortran etc., are executed in a sequential order.

\[
\text{BOUNDARY. } U = 0.0 \text{ ON } \begin{cases} 
X = 1. - \cos(\pi*\text{THETA}), & \\
Y = 1. - \sin(\pi*\text{THETA}) & \\
\text{FOR } \text{THETA} = 0. \text{ TO } 2.
\end{cases}
\]

Figure 3.3.: Domain definition and boundary condition assignment in ELLPACK.

The PDE and the boundary conditions written in the PDE language are symbolically processed using the Macsyma [24] symbolic system. Procedural (Fortran) code is generated and linked with the selected libraries and executed. The PELLPACK system contains libraries of modules for the different steps of the solution process, such as domain discretization modules and PDE solver libraries. There are several integrated libraries that are ready to use in the system. It is also possible to integrate new libraries by writing the appropriate interfaces on different levels of the software architecture.

3.2.3. PDESpec

S. Weerawarana [46] presents a high-level language PDESpec where the PDE problem is specified using objects. Different types of objects are equation, do-
Chapter 3. Related Work

main, boundary_condition, initial_condition, mesh, decomposition, algorithm, solve and solution. Figure 3.4 shows an example of an equation object describing the steady-state heat flow, where \( T_x \) represents \( \partial_x T \) and \( D_x(A) \) is an alias for \( \text{diff}(A, x) \) which represents \( \partial_x(A) \). Aliases, the dimension of the problem, and names of the independent variables are defined as defaults for equation objects. The domain “dome” is defined as a separate object, as well as the boundary conditions and their equations. The definition of the domain object can be seen in Figure 3.5.

```plaintext
equation (  
  name = "steady-state heat flow",  
  domain = "dome",  
  expressions = [ Dx( k(x,y)*Tx ) + Dy( k(x,y)*Ty ) = 0 ],  
  properties = [ [self-adjoint], [steady-state] ]
);
```

Figure 3.4.: An equation object in PDESpec describing the steady-state heat flow.

```plaintext
domain (  
  name = "dome",  
  type = piecewise_parametric,  
  boundary = [    
    orientation = clockwise,    
    parametric (x=3, y=.7-t, t, 0, .7),    
    ...    
  ]
);
```

Figure 3.5.: Definition of a domain object in PDESpec.

Besides PDESpec, a problem solving environment with an extensible architecture for different solvers and an intelligent PDE solver selection based on expert system methodology are presented.

3.2.4. FEMLAB

FEMLAB [13] is a Matlab package for solving PDE problems. Both steady-state and time-dependent problems can be solved, as well as scalar PDEs and systems of PDEs. An environment with a graphical user interface as well as the Matlab command line interface is used for problem specification, solution, and visualization. Figure 3.6 shows an example of the FEMLAB user interface, where surfaces of the three-dimensional domain can be individually highlighted and assigned different boundary conditions.
3.2. High Level Packages and Problem Solving Environments

Figure 3.6.: Surface selection in the FEMLAB graphical user interface, useful for example during boundary condition specification.

Figure 3.7.: Visualization of the PDE solution in FEMLAB. The solution shows an example from structural mechanics, more specifically the static deformation of a feeder clamp.
Chapter 3. Related Work

FEMLAB provides different modes, each with several built-in models. In the PDE Mode, the coefficients of a predefined PDE are specified in coefficient form or in general form suited for non-linear problems. In the Physics Application Mode, the model can be described by its physical parameters rather than through its PDE coefficients. There are several built-in models to choose from in a number of application areas like electromagnetics, heat transfer, structural mechanics etc. There is also support for multiphysics problems, where, for instance both electric and thermal effects can be studied simultaneously.

Once a model has been selected, the domain geometry is defined using a built-in two- or three-dimensional geometry editor with predefined geometric objects that can be combined to build complex geometries. Then the PDE and boundary conditions are assigned to each part of the domain and its boundary. Next, a finite element mesh is automatically generated, which can be viewed and refined interactively. Finally, the problem is solved and the result can be visualized and analyzed, see Figure 3.7.

3.3. Discussion

In the beginning of this chapter we classified available systems for solving PDEs into two groups:

- programming language based packages,
- high-level packages.

The first category of tools is directed towards users with numerical knowledge and some programming language skills. Also, a varying degree of manual work on the solution must be done before the actual implementation. On the other hand, an efficient solver can be obtained for a specific problem.

The second category contains tools that do not require numerical knowledge to the same extent, and the details of the solution process are hidden. This is true especially for the problem solving environments, and in particular FEMLAB. In FEMLAB, existing models can be used directly, with some modification of the parameters and new models can be defined using the graphical user interface. Although, the language support is poor, model parameters can be set directly using the Matlab interface and there is a number of help functions. PELLPACK has a more advanced, but still limited, specification language. The tool with the most advanced language is gPROMS, which defines a formal modeling language and supports hierarchical modeling, domain description, etc., but only rectangular domains are supported.
PDEModelica belongs to the second category, being a high-level language, and aims to hide numerical details from the user. The closest tool to PDEModelica is gPROMS which only supports simpler domains. Compared to the other packages PDEModelica has additional object-orientation support, like inheritance, as well as support for a general connection concept.
Chapter 3. Related Work
Chapter 4.
PDEModelica

This chapter defines the problem to be solved in this thesis and presents a possible solution. The solution is a set of language extensions to the Modelica language, consisting of constructs for component-based domain geometry description, operators to express partial differential equations in the language, and constructs for defining a complete PDE model containing the domain, the equations and the boundary conditions.

4.1. Problem Statement

The purpose of this work is to examine whether a general, object-oriented, declarative modeling language with support for ordinary, algebraic, and partial differential equations as well as component based modeling can be defined. Another goal is to design language constructs for this purpose, based on the existing modeling language Modelica, and to evaluate whether the extended language fulfills the given requirements. The problem is to design language constructs that allow a modeler to describe domain geometries, boundary conditions and partial differential equations in a language using object orientation and component-based, hierarchical modeling. Part of this problem also involves the definition of a connection concept for PDE-based models and a connection mechanism for connecting ODE- and PDE-based models.

4.2. PDE Problem Specification

The following parts are specified for describing a PDE problem:

- the geometry of the solution domain,
- the partial differential equation,
Chapter 4. PDEModelica

- the boundary conditions, and

- the initial conditions in case the PDE problem is time-dependent.

The computed solution is a function of space for a stationary problem, and a function of space and time for a time-dependent problem.

4.3. Domain Geometry Definition

The domain of a PDE problem is $D \subset \mathbb{R}^n$. In this work we mainly consider the two-dimensional case, $n = 2$. In order to define the domain a geometric region must be described. This can be done by describing the boundary of the geometric region or by combining previously defined regions into new regions.

4.3.1. Boundary Description

In most practical cases it is sufficient to define the domain by a parametric curve $\{(x_s, y_s) \mid s \in [s_{\text{start}}, s_{\text{end}}]\}$ describing the boundary of the region, which is a sufficiently general way of stating the geometry of the domain. In the case of more complex geometries, the boundary can be divided into several parts and each part can be described by a separate parametric curve. The complete curve should be closed and not be 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 on the left side of the curve in the forward direction with respect to the parameter. A domain class can also be used to define a partial, i.e. non-closed boundary that can be used in another domain as part of a complete boundary.

**The boundary Section**

The description of the domain in a domain restricted class is specified by a boundary section, identified by the keyword `boundary`. The `boundary` section is a special case of the general Modelica equation section to specify equations that constrain the space coordinates to be on the domain. When the domain restricted class defines a boundary, the `boundary` section constrains the space coordinates to be on the boundary.

For convenience boundary description operators are provided. Currently, three boundary description operators are defined in PDEModelica, `line()`, `curve()`, and `composite()`.
4.3. Domain Geometry Definition

**The line() Operator**

This operator specifies a list of two-dimensional coordinates that define a list of connected lines, called a polyline. Each pair of consecutive points in the list define a line, building a continuous curve with corners at the connection points, see Figure 4.1. The resulting lines must not be self-intersecting. If the domain class with the boundary described by `line()` is to be used as the domain geometry in a PDE problem, the boundary must be closed, i.e. the start and the end points must have the same coordinates.

\[
\begin{align*}
(x_0, y_0) & \quad (x_1, y_1) & \quad (x_n, y_n) \\
(x_1, y_1) & \quad (x_2, y_2) & \quad (x_{n-1}, y_{n-1}) \\
\end{align*}
\]

Figure 4.1.: A set of connected lines that is described by the `line()` construct `line(\{(x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n)\})`

The simplest case is a single line defined by its start and end points. Figure 4.2 shows the PDEModelica code that defines a `Line2D` class.

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

Figure 4.2.: Definition of the Line2D class using the `line()` operator

**The curve() Operator**

The `curve()` operator specifies a parametric expression that defines the boundary or boundary part. One parameter is needed in the parametric expression for a curve in two-dimensional space. The parameter is assumed to be in the interval \([0, 1]\). The `curve()` operator contains one expression for each coordinate, i.e. one for each of the \(x\) and \(y\) coordinates in two dimensions. As with the `line()` operator, in order to be used to specify the complete domain geometry of a model, the curve must be not be self-intersecting and should be closed.

Figure 4.3 shows how a `Bezier2D` class can be defined which can be used to represent boundary parts using Bézier curves. Bézier curves \([8, 9]\) are parametric curves defined by a number of control points. Position of each
point on the curve is calculated by a weighted sum of all the control points, where the weights are functions of the position on the curve varying between 0 and 1. One simple way of calculating the points of the curve is using de Casteljau’s algorithm [9,11], which is implemented in the function 

```
function bezierfunc
  input Real px[:];
  input Real u;
  output Real res;
  Real qx[size(px,1)];
algorithm
  qx := px;
  for k in 1:size(px, 1) - 1 loop
    for i in 1:size(px,1) - k loop
      qx[i] := (1-u)*qx[i] + u*qx[i+1];
    end for;
  end for;
  res := qx[1];
end bezierfunc;
```

When declaring an instance of the Bezier2D class in Figure 4.3, two arrays must be given as parameters, one array containing the \(x\)-coordinates of the control points and one containing the \(y\)-coordinates.

```
domain Bezier2D "Boundary domain"
  extends Cartesian2D;
  parameter Real px[:];
  parameter Real py[size(px,1)];
  parameter Real u;
boundary
  curve(bezierfunc(px, u), bezierfunc(py, u));
end Bezier2D;
```

Figure 4.3.: Definition of the Bezier2D class using the `curve()` operator.

**The composite() Operator**

This operator defines a composite boundary from a list of boundary components declared in the domain class it resides. The boundary components are instances of domain classes defined separately, which in turn are defined using one of the three operators. The boundary components must be listed in the correct order so that the end point and the start point of consecutive components have the same coordinates. The end point of the last component and the start point of the first component must also have the same coordinate so that the complete curve is closed, if the domain is to be used to define the geometry of a PDE problem.
4.3. Domain Geometry Definition

Figure 4.4 shows an example of a composite domain Rectangular2D consisting of four sides declared as separate boundary components. The domain is illustrated in Figure 4.5.

```plaintext
domain Rectangle2D "Two-dimensional domain"
  extends Cartesian2D;
  parameter Real cx=0, cy=0, w=1, h=1;
  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;
```

Figure 4.4.: Definition of the Rectangle2D class using the composite() operator.

Figure 4.5.: The domain defined by the Rectangle2D class in Figure 4.4.

4.3.2. Complex Domains

Many PDE problems are based on physical systems consisting of several different physical materials with different properties, see for example Figure 4.6. When defining such problems the complete domain is the union of the subdomains defined for each different material. Parts with different materials can be seen as different components that are combined together to build the final system. Such domains can be defined using aggregation, declaring subdomains as components and combining them using operators.

Boolean operators such as union, intersection and difference can be used to combine domains. Using difference, subdomains can be defined to represent holes in the domain. Furthermore, the connector concept can be generalized to several dimensions in order to be used for component based modeling with multidimensional objects, see Section 7.2.2. Using union or
connector operators, models such as the static current problem in Section 6.2 can be specified.

\begin{verbatim}
aluminum

copper

aluminum
\end{verbatim}

Figure 4.6.: Example of a complex domain consisting of three subdomains with two different physical materials.

In our current work, building really complex geometric domains has not yet been investigated.

4.3.3. Domain Classes and Boundary Classes

There is a need to separate two different kinds of domain classes:

- classes that define the boundary of a domain, and
- classes that define the domain itself.

The boundary classes have one dimension less than the domain classes, but they still need to be defined in the same number of dimensions. The current implementation does not distinguish these two kinds in the language. Instead, if the composite() operator is present in a domain class, that domain class is assumed to define the domain itself, not just a composite boundary.

For example, the boundary based description in Section 4.3.1 describes the line() and the curve() operators which define one-dimensional regions distributed in two-dimensional space, whereas the current implementation of the composite() operator defines two-dimensional domains in two-dimensional space.

The domain classes use instances of the boundary classes to define a domain. See Figure 4.7 for an example. A domain class using the curve() operator defines a one-dimensional domain, a curve, in two-dimensional space, as seen in Figure 4.7 (a). When an instance of this class is used in another domain with the composite() operator, a two-dimensional domain is defined, as seen in Figure 4.7 (b). In order to define complex domains, domain classes must be able to instantiate two dimensional domains and combine them to define a complex domain. Further investigation and design is needed to support and clarify this issue in PDEModelica.
4.4. Model Definition

This section discusses the definition of the PDE-based model in terms of the formulation of the partial differential equation itself and the associated boundary conditions. A coefficient based approach is described where a predefined general PDE is assumed and only the coefficients remain to be defined, as well as new operators that are required for conveniently expressing PDEs in PDEModelica.

4.4.1. Operators

In order to represent the equations of the model, the initial conditions and the boundary conditions, new operators need to be introduced in PDEModelica. The basic operator is the new partial derivative operator for derivatives with respect to space variables, analogous to the \( \text{der}() \) operator in Modelica that represents time derivative of variables.

An operator for the normal derivative on a domain boundary is also needed for certain boundary conditions.

Moreover, special derivative operators such as divergence and gradient are often used in mathematics to more conveniently express certain PDEs.

Partial Derivative

For partial derivatives of the form \( \frac{\partial^m}{\partial x^m} (expr) \), a natural choice is to use the operator \( \text{pder}(expr,x,m) \), where \( \text{expr} \) is an expression and \( x \) is a space variable or the time variable, and \( m \) is the differentiation order that by default is one if omitted. The \( \text{der}() \) operator in Modelica is equivalent to \( \text{pder}(expr,time) \) with \( \text{time} \) being the builtin time variable in Modelica.
Chapter 4. PDEModelica

Divergence and Gradient

A short and very common mathematical notation for expressing PDE models involves the gradient and the divergence operators, $\nabla$ and $\nabla \cdot$ respectively. If a Cartesian coordinate system in two-dimensional space is used, the gradient operator is defined as

$$\nabla u = \left( \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right)$$

Hence, the gradient operator increases the dimensionality of its argument. On the other hand, the divergence reduces the dimensionality of its argument. With the same coordinate system, and using the argument

$$u = \left( u_1, u_2 \right)$$

the divergence operator is defined as

$$\nabla \cdot u = \partial_x u_1 + \partial_y u_2$$

Thus, using these operators together, the Laplace equation can be expressed as

$$\nabla \cdot (\nabla u) = 0$$

which given the same coordinate system as above, is equivalent to

$$\partial_{xx} u + \partial_{yy} u = 0$$

These operators can be introduced in PDEModelica, `divergence()` for the divergence operator and `gradient()` for the gradient operator. The advantage of also introducing these operators besides the `pder()` operator is that the syntax of these special operators is independent of the dimensionality of the problem and the names of the coordinates unlike the `pder()` operator.

Directional Derivative

In some boundary conditions, the outward normal derivative on the domain boundary occurs, usually written $\frac{\partial^m}{\partial \nu^m}(expr)$, most commonly only being the first-order derivative, $m = 1$. The outward normal derivative is the directional space derivative in the outward normal direction, see Figure 4.8. Since the normal vector depends on the domain geometry, this derivative cannot be represented by the `pder()` operator alone. Either the normal vector must be available or a special operator must be used to represent the normal derivative, for example the operator `nder(expr,m)`, with `nder(expr)` corresponding to $m = 1$. 

38
4.4. Model Definition

Outward Normal Vector

The normal derivative can also be expressed by a scalar product between the normal vector and the gradient. If the formulation using the gradient operator is preferred also when the boundary conditions are specified, the outward normal vector needs to be made available. This can either be a built-in component or a keyword in the language that is automatically replaced by the normal vector during translation since the domain will not be available during the formulation of the boundary condition; this is done separately from the domain definition.

\[
\frac{\partial}{\partial n} \partial \frac{\partial}{\partial x} \partial \frac{\partial}{\partial y}
\]

Figure 4.8.: The normal vector on the boundary that is used in the normal derivative operator. The normal derivative is the directional space derivative in the normal direction.

4.4.2. Hierarchical Definition of PDEs and Boundary Conditions

In order to simplify PDE model definition, a general PDE model can be expressed as a base model in PDEModelica 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 that can subsequently be instantiated and used when needed.

Analogously, boundary conditions can be defined using base models and inheritance. A coefficient-based PDE base model can be defined as in Figure 4.9. The variable \( u \) represents the unknown variable, which is 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 equation in PDECoefficient2D expressed in mathematical notation:

\[
da \frac{\partial u}{\partial t} - \nabla \cdot (c \nabla u) + au = f
\]
Chapter 4. PDEModelica

```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;
```

Figure 4.9.: A coefficient-based PDE base class.

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

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

This represents the Poisson equation. A general, time-dependent `HeatTransfer` model with coefficients common for heat transfer problem formulations can be defined as follows:

```model HeatTransfer
  extends PDECoeff2D(da=rho*C, c=k, a=h, f=Q+h*T_ext);
  parameter Real rho=0 "density";
  parameter Real C=0 "heat capacity";
  parameter Real k=0 "thermal conductivity";
  parameter Real h=0 "convection coefficient";
  parameter Real T_ext=0 "External temperature";
end HeatTransfer;
```

This model represents the following equation:

\[
\rho C \frac{\partial u}{\partial t} - \nabla \cdot (k \nabla u) = Q + h(T_{ext} - u)
\]

with \( u \) being the temperature, i.e. the solution. Hence, time dependency is chosen by modifying the coefficient of the time derivative term.

**Boundary Conditions**

A Robin boundary condition, used in a heat transfer problem to describe a boundary that is neither a perfect conductor nor a perfect insulator, can be
stated by first writing a general Robin boundary condition, as seen in Figure 4.10. In mathematical notation this equation is formulated as follows:

\[ c \frac{\partial}{\partial n} u + qu = g \]

Other types of boundary conditions, e.g. Dirichlet and Neumann conditions, can be used to describe a perfect heat conductor and a perfect insulator, respectively. Both of these boundary conditions can be defined by extending the \texttt{Robin} class and setting the appropriate parameters to zero, for example the \texttt{Neumann} and \texttt{Dirichlet} boundary condition models in Figure 4.11.

Regarding heat transfer problems, a more specific version of the Robin boundary condition can be defined by inheriting the \texttt{Robin} class, adding application specific parameters, and mapping them to the general parameters, see Figure 4.12. The corresponding mathematical equation with these parameters is

\[ \frac{\partial}{\partial n} (k \nabla u) = qh + hh(T_{\text{ext}} - u) \]

where \( qh \) is the source term, \( hh \) is the heat transfer coefficient and \( T_{\text{ext}} \) is the external temperature.
Chapter 4. PDEModelica

model HeatRobin "For heat transfer"
  extends Robin(c = k,
    q = hh,
    g = qh + hh*T_ext);
  parameter Real k = 1;
  parameter Real qh = 0 "Heat flux";
  parameter Real hh = 1 "Heat transfer coefficient";
  parameter Real T_ext = 25 "External temperature";
end HeatRobin;

Figure 4.12.: Heat transfer specific model derived from the general Robin model.

4.4.3. Complete Model Definition

Once the model classes for the PDE and the boundary conditions have been formulated 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 setting the instance equal to the variable eq (short for equation), also a built-in variable of the restricted class domain.

The complete problem statement can be seen in Figure 4.13. 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. A Robin boundary condition is used for a non-isolating glass layer on the left side, and a Neumann boundary 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.

With time-dependent problems, initial conditions must be supplied. The Real type in Modelica has an attribute called start for this purpose. Also, an initial equation section can be defined as of Modelica version 2.0 [27], as follows:

model PDEModel
  Rectangle2D dom;
  HeatTransfer ht(u(start=0)); // first alternative
  initial equation
    ht.u = 0; // second alternative
  equation
    dom.eq = ht;
end PDEModel;
4.5. Summary of Extensions in PDEModelica

A summary of the introduced extensions in PDEModelica follows:

- **space**: For declaring space variables to represent space coordinates.
- **domain**: New restricted class.
- **boundary**: New section in domain restricted classes for domain geometry definition.
- **curve()**: Boundary description using parametric curves.
- **line()**: Boundary description using straight lines.
- **composite()**: Boundary geometry description composed by other boundary segments.
- **nder()**: Outward normal derivative operator.
- **pder()**: Partial derivative operator.
- **divergence()**: Divergence operator.
- **gradient()**: Gradient operator.

Figure 4.13.: Complete specification of a PDE problem using previously defined domains, PDE and boundary condition models.

Initial values that are not constant over the domain can be expressed by defining a function. However, this has not been tested yet in the current work and needs to be further investigated.
Chapter 4. PDEModelica

- **eq**: Predefined variable in `domain` restricted classes for assigning PDEs to domains.

- **bc**: Predefined variable in `domain` restricted classes for assigning boundary conditions to boundary segments.
Chapter 5.
Implementation

The PDEModelica prototype consists of several modules. The input model specification to the system is a text file containing PDEModelica code. PDEModelica is the Modelica translator Modeq with PDE extensions which extracts domain information and PDE parameters from the input source. From the domain definitions in PDEModelica, a discrete domain description is generated by the domain discretizer for the specific mesh generator used. The mesh generator and the PDE solver can be external software modules freely available with source code. Figure 5.1 provides an overview of the system.

The Modelica translator consists of several parts. The first step is the lexer and the parser that reads textual Modelica code and generates an abstract syntax tree. Next, a generated compiler interprets the code and translates the models into a flat set of differential and algebraic equations and possibly a set of functions. A numerical solver is then used to solve the set of equations and simulate the models. The different steps of the translation process and the tools used can be seen in Figure 5.2.

The PDE extensions are described separately, in sections Section 5.1.1 and Section 5.2.3. This implementation uses the coefficient-based approach, assuming that a certain general equation is used, and only handles the parameters of the PDE model, ignoring the equations sections. Thus, the special operators for defining PDEs are not mentioned. Future work regarding these issues is discussed in Chapter 7.

5.1. Modelica Parser

The Modelica lexer and parser are implemented using ANTLR [4], a translator generator with special meta-languages for grammar specifications. The lexer grammar specification consists of definition of the tokens, special characters and comments. The reserved keywords of the language are listed in a tokens section:
Chapter 5. Implementation

Figure 5.1.: Overview of the PDEModelica prototype.

Figure 5.2.: The different steps of the translation process, the tools used and the intermediate results.
5.1. Modelica Parser

tokens {
  ...
  CLASS = "class" ;
  CONNECT = "connect" ;
  CONNECTOR = "connector" ;
  ...
}

The new keywords introduced in PDEModelica compared to Modelica are domain, boundary, line, curve, composite and space.

Operators, special characters, comments and literals are defined using rules. Each rule consists of a name and a definition, and optionally actions to be performed when the rule is matched. The syntax for a rule is:

```
rule_name
  : alternative_1
    | alternative_2
    ...
    | alternative_n
  ;
```

Each alternative contains a rule that is checked for a match. Table 5.1 lists some of the symbols used in ANTLR rules.

Table 5.1.: Some symbols in ANTLR used in rules, consisting of regular expressions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(...)</td>
<td>Subrule</td>
</tr>
<tr>
<td>(...)*</td>
<td>Closure subrule (repeat zero or more times)</td>
</tr>
<tr>
<td>(...)+</td>
<td>Positive closure subrule (repeat one or more times)</td>
</tr>
<tr>
<td>(...)?</td>
<td>Optional</td>
</tr>
<tr>
<td>{...}</td>
<td>Semantic action</td>
</tr>
<tr>
<td></td>
<td>Alternative operator</td>
</tr>
<tr>
<td>..</td>
<td>Range operator</td>
</tr>
<tr>
<td>.</td>
<td>Wildcard</td>
</tr>
</tbody>
</table>

Rules are also used in the parser, but instead of characters a stream of tokens is the input that is being parsed, and a parse tree is automatically generated.

ANTLR also supports the implementation of tree walkers using a meta-language similar to the lexer and parser definition meta-languages. Tree walkers can be used to traverse a syntax tree built by the parser and perform some actions or transformations on the tree.
Chapter 5. Implementation

5.1.1. PDEModelica Extensions

Some additions have been done to the Standard Modelica parser to handle the PDE extensions. New keywords and operators are summarized in Table 5.2. `domain` is a new kind of restricted class to describe domain geometry, with a new section called the `boundary` section. The boundary section is described by the rules shown in Figure 5.3.

Table 5.2.: New keywords and operators in PDEModelica.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>Space variables for coordinates</td>
</tr>
<tr>
<td>domain</td>
<td>New restricted class <code>domain</code></td>
</tr>
<tr>
<td>boundary</td>
<td>New section in classes for domain geometry definition</td>
</tr>
<tr>
<td>curve(), line(), composite()</td>
<td>Boundary geometry description operators</td>
</tr>
<tr>
<td>eq.bc</td>
<td>Predefined variables in <code>domain</code> classes</td>
</tr>
</tbody>
</table>

```
boundary  : 
    | line_clause
    | curve_clause
    | composite_clause
   )
   comment!

line_clause  :
   LINE^ LPAR! e:expression RPAR!

curve_clause  :
   CURVE^ LPAR! e:expression_list RPAR!

composite_clause  :
   COMPOSITE^ LPAR! el:component_reference_list RPAR!
```

Figure 5.3.: Parser rules in ANTLR for the boundary section.

For description of the operators, see Section 4.3.1.

All three pseudo functions use a syntax with parentheses similar to the connect operator, represented by LPAR and RPAR. The symbols "\(^\)" and

48
5.2. Modelica Translator

'!' are used to control the automatic parse tree generation, '"' defines a
node to be the root node of a current subtree, and '!' suppresses the node
from the subtree. Figure 5.5 shows the abstract syntax tree resulting from
the PDEModelica domain example in Figure 5.4.

```modelica
domain TestDomain
  space Real x,y;
  parameter Real PI = 3.141593;
  parameter Real u;
  boundary
    curve(cos(2*PI*u), sin(2*PI*u), u);
end TestDomain;
```

Figure 5.4: A domain example in PDEModelica.

Figure 5.5: The abstract syntax tree for the domain class example in Fig-
ure 5.4. The expression subtrees are compressed into single
strings for readability.

5.2. Modelica Translator

A tree walker written in ANTLR traverses the abstract syntax tree gen-
erated by the parser and creates a corresponding tree using the data types
Chapter 5. Implementation

declared in RML. The tree is subsequently processed further by the Modelica
translator described in the following sections.

5.2.1. RML

RML (Relational Meta Language) [14, 33] is a meta language for formal
specification of language semantics using Natural Semantics [21]. A transla-
tor for the language described in RML is automatically generated using the
rm12c compiler, which generates efficient C code that is compiled into an
executable.

Specifications in Natural Semantics are expressed using inference rules
similar to natural deduction used in logics. Inference rules with the same
input and output formal parameters can be grouped into signatures. Each
rule is defined by a set of propositions called premises and a proposition
called conclusion. A proposition is proved by applying a rule with a conclu-
sion that can be instantiated to that proposition, and the premises of that
rule are proven similarly. An inference rule with the premises $p_1..p_n$ and the
conclusion $p$ is written

$$p_1 \land p_2 \land \cdots \land p_n \quad \Rightarrow \quad p$$

In RML, signatures are defined using relations. Each relation consist of
a set of rules. Rules contain propositions consisting of relation invocation
with arguments and results separated by the $\Rightarrow$ symbol. The premises and
the conclusion are separated by a line with two or more consecutive dashes,
see Figure 5.6. The syntax is otherwise similar to that of Standard ML [26].
An example relation can be seen in Figure 5.6.

The first part of an RML specification file contains the data type dec-
larations and relation declarations that are visible to other modules. The
module exprmod in Figure 5.6 contains one data type, Exp, that can be ei-
ther an integer constant or an addition operation with two subexpressions.
Integer numbers are represented by the primitive type int. The only rela-
tion in this module is called eval, which evaluates the value of an expression
with the integer value as the result.

Rules without premises are called axioms, as in the first rule of the eval
relation, which returns the value of an integer constant. The second rule
handles the addition operations by first evaluating the subexpressions and
adding their values using the built-in RML operator int_add. The result of
the rule is then the result of that addition.
module exprmod:
    datatype Exp = INTCONST of int |
    OP_ADD of Exp * Exp
    relation eval : Exp => int
end

relation eval : Exp => int =
axiom eval INTCONST(v) => v
rule eval (e1) => v1 &
    eval (e2) => v2 &
    int_add (v1, v2) => v3
    ---------------------
    eval (OP_ADD (e1, e2)) => v3
... end

Figure 5.6.: Example of an RML specification. The first part contains the
data type definitions and the module interface definition. The
second part contains the relations. \texttt{int} is the built-in integer
type, and \texttt{int_add} is the built-in integer addition operator.

5.2.2. Modelica Translator Specified in RML

A partial semantic specification of an early version of Modelica was initially
developed by David K˚ agedal \cite{22,23} in RML, and has later been substantially
expanded and improved. This specification describes the static semantics of
the Modelica language, i.e. the interpretation of the object-oriented struc-
ture, typing, and declarations defined in the models. The dynamic semantics
involving the run-time simulation of the models is not included. Hence, this
specification describes the translation of Modelica models with structural
and compositional information into a flat list of equations.

The translator is divided into several modules. The \texttt{Absyn} module con-
tains the representation of the abstract syntax tree, with data types that
represent classes, variable declarations, equations, algorithms, expressions,
and statements. The \texttt{Absyn} representation is close to the source representa-
tion of the Modelica code, i.e. the source can be recreated from the abstract
syntax.

From the abstract syntax tree the \texttt{SCode} intermediate tree is generated.
This tree is defined in the \texttt{SCode} module. The \texttt{SCode} tree is a canonical
representation of the Modelica code for more convenient translation. For
example, the Modelica code and consequently the abstract syntax tree can
contain multiple equation sections, depending on how the code was entered.
In the \texttt{SCode} representation the equation sections are merged into one list.
Chapter 5. Implementation

of equations that is semantically equivalent and makes further processing simpler.

From the SCode representation, the appropriate Modelica class is instantiated by the Inst module. The result of the instantiation is the DAE representation defined in the DAE module, which contains the resulting functions and a flat set of equations resulting from the translation. Instantiating a Modelica class means:

- retrieving the class definition of the extended classes if there are any extends clauses and inserting their contents into the current model,
- retrieving the class definition of the declared class of each component in the model,
- instantiating their contents with the current component name prefix to avoid name clashes,
- applying the modifications and inserting their contents into the current model,
- building connections sets and generating equations from these, and
- merging the equations from the instantiated subcomponents updated with the new component names.

During the instantiation, some type checking and constant expression evaluation are also performed. Type checking also includes checking whether the subtype relationship holds between Modelica classes. Figure 5.7 shows the RML relation that checks subtyping and type equivalence as defined in Section 2.1.

5.2.3. PDEModelica Extensions

In this section the additions to the Modelica translator to handle domain restricted classes and PDE parameters are described. These are additional data types in the RML representation of the abstract syntax tree, and rules for instantiation of domains and code generation for domains and PDE parameter initialization.

Representation

Several extensions have been in different parts of the Modelica semantics specification to represent and translate PDEs and domains. In the Absyn module, the Restriction data type is extended with the R_DOMAIN and R_PREDEFINED_DOMAIN. The ClassPart data type is modified to support a
relation subtype : (Type, Type) => bool =

axiom subtype (T.INTEGER, T.INTEGER) => true  
axiom subtype (T.REAL, T.REAL) => true  
axiom subtype (T.STRING, T.STRING) => true  
axiom subtype (T.BOOL, T.BOOL) => true

... 

rule subtype_varlist(el1, el2) => true 
-----------------------------------
    subtype(T_COMPLEX(st1, el1), T_COMPLEX(st2, el2)) => true 

axiom subtype(t1, t2) => false
end

relation equivtypes : (Type, Type) => bool =

rule subtype(t1, t2) => true &
    subtype(t2, t1) => true  
-----------------------
    equivtypes(t1, t2) => true 

axiom equivtypes(t1, t2) => false
end

Figure 5.7.: The RML relations that check subtyping and type equivalence between to Modelica classes. Subtyping rules regarding arrays and tuples are not shown. The relation subtype_varlist calls the subtype relation for the declared types of each pair of variables in the given variable lists.
Chapter 5. Implementation

boundary node with a list of boundary elements, where each element can be one of the three boundary types, see Figure 5.8. In the current prototype, the representation is kept simple, using the Exp data type for the LINES node, which corresponds to the list of points written as a single expression using the array syntax. Similarly, the CURVE node contains a list of expressions, one for each of the coordinates.

```plaintext
datatype ClassPart = PUBLIC of Element list |
                     | PROTECTED of Element list |
                     | EQUATIONS of Equation list |
                     | ALGORITHMS of Algorithm list |
                     | BOUNDARY of Boundary list |

datatype Boundary = LINES of Exp |
                     | CURVE of Exp list |
                     | COMPOSITE of ComponentRef list |
```

Figure 5.8.: Data type extensions for domain representation in the Absyn module.

The representation of boundaries in the SCode module are the same as in the Absyn module, but the ClassDef data type has been modified to represent domain information as well (see Figure 5.9).

```plaintext
datatype ClassDef = PARTS of Element list |
                    | DERIVED of Path |
                    | | ABSYN.ArrayDim option |
                    | | Mod |
                    | | DOMAINPARTS of Element list |
                    | | BOUNDARY of Boundary list |
```

Figure 5.9.: Data type extensions for domain representation in the SCode module.

The DAE representation is also extended with nodes needed to represent the extra information related to PDEs, see Figure 5.10. Data types for expressions stored in the DAE representation are defined in the Exp module, and contain the statically analyzed expressions, as opposed to the Absyn.Exp data type which contains the expressions before the analysis. The DAEList stored in the CURVE node is needed during code generation to determine which component references in the curve expressions that are parameters, in order to prefix them with the correct prefix. See Section 5.2.4 for description of the code generation phase.
5.2. Modelica Translator

Figure 5.10.: Additions in the DAE module for storing domain information.

The only addition in the Element data type is the DOMAIN node.

Translation

The SCode module contains the relations that convert the Absyn representation to the SCode representation, beginning with the elaborate relation. Some relations in this module are modified to handle the domain extensions. The elab_class relation redirects the translation to elab_domaindef instead of elab_classdef if the current class has the restriction type R_DOMAIN, i.e. if the class being translated is a domain class. Conversion of the declaration part in elab_domaindef is identical to the conversion of other classes, but the conversion of equation and algorithm sections are replaced with a relation that converts the boundary section. The result of the boundary section conversion is the SCode.DOMAINPARTS structure. The first part of this structure are the elements from the declaration part, and the second part is the boundary description consisting of SCode.Boundary elements generated by the relation elab_boundary_parts, see Figure 5.11.

The translation is done in the Inst module in different ways depending on the type of the SCode.ClassDef node, one of PARTS, DERIVED or DOMAINPARTS. In the latter case, the rule for translating the domains is activated. The declaration part is handled as for other classes. The boundary part is instantiated using the relation inst_boundary, see Figure 5.12. In this relation the three different kinds of boundary descriptions POINTS, CURVE and COMPOSITE are recognized and corresponding DAE nodes are generated. The common steps for each kind is:

- **static analysis**: constant expression evaluation, type analysis and type annotation of expressions, variable references, etc. In the COMPOSITE

```plaintext
datatype DAElist = DAE of Element list

datatype Element = VAR of Exp.ComponentRef * VarKind * VarDirection * Type * Exp.Exp option |
  DEFINE of Exp.ComponentRef * Exp.Exp |
  EQUATION of Exp.Exp * Exp.Exp |
  ALGORITHM of Algorithm.Algorithm |
  COMP of Ident * DAElist |
  FUNCTION of Absyn.Path * DAElist * Types.Type |
  DOMAIN of Exp.ComponentRef * Boundary list

datatype Boundary = LINES of Exp.Exp * Static.Properties |
  CURVE of Exp.Exp list * DAElist |
  COMPOSITE of Exp.Exp list
```

```plaintext
Figure 5.10.: Additions in the DAE module for storing domain information. The only addition in the Element data type is the DOMAIN node.
```
relation elabbndparts: Absyn.Boundary list => Boundary list =

    axiom elabbndparts ([ ]) => [ ]

    rule elabbndparts (rest) => bnds &
        let bnds' = POINTS(e)::bnds
        ------------------------
        elabbndparts (Absyn.POINTS(e)::rest) => bnds'

    rule elabbndparts (rest) => bnds &
        let bnds' = CURVE(elist)::bnds
        ------------------------
        elabbndparts (Absyn.CURVE(elist)::rest) => bnds'

    rule elabbndparts (rest) => bnds &
        let bnds' = COMPOSITE(elist)::bnds
        ------------------------
        elabbndparts (Absyn.COMPOSITE(elist)::rest) => bnds'

end

Figure 5.11.: The relation `elab_boundary_parts` (abbreviated to elabbndparts) in the SCode module that handles the boundary description parts from the Absyn module.

In the current instantiation environment. A reference to a variable v in a variable `comp` in the class being instantiated is prefixed by `comp.` to build `comp.v`.

- **state transition in class inference**: modifying the state in the class inference module `ClassInf` which keeps track of the class state in order to check if the class contents is consistent with the restricted class type. In this case the transition to `FOUND_BOUNDARY` has been added.

### 5.2.4. Code Generation

The result of the Modelica translation phase is the DAE representation that contains the list of variables, parameters, functions, equations, algorithms, and domains. In this prototype, there is also a code generation module for setting up the PDE problem in C++ code so that it can be compiled and linked with a PDE solver. The code generation phase produces code for
5.2. Modelica Translator

relation inst_bnd:(Env, Mod, Prefix, Connect.Sets,
    ClassInf.State, SCode.Boundary, DAE.DAElist)
    => (DAE.Boundary list, Env, Connect.Sets, ClassInf.State) =

    rule Static.elab_exp(env, e) => (e', prop) &
        PrefixCode.prefix_exp (env, e', pre) => e'' &
        ClassInf.trans(st, ClassInf.FOUND_BOUNDARY) => st'
        ---------------------------------------------------------------
        inst_bnd(env, _, pre, csets, st, SCode.POINTS(e), dae)
        => ([DAE.POINTS(e'', prop]), env, csets, st')

    rule Static.elab_exp_list(env, elist) => (el', _) &
        PrefixCode.prefix_exp_list (env, el', pre) => el'' &
        ClassInf.trans(st, ClassInf.FOUND_BOUNDARY) => st'
        ---------------------------------------------------------------
        inst_bnd(env, _, pre, csets, st, SCode.CURVE(elist), dae)
        => ([DAE.CURVE(el'',dae)], env, csets, st')

    rule Static.elab_cref_list(env, crlist) => (el',_,_) &
        canon_cref_list(env, el') => el'' &
        PrefixCode.prefix_exp_list (env, el', pre) => el''' &
        ClassInf.trans(st, ClassInf.FOUND_BOUNDARY) => st'
        ---------------------------------------------------------------
        inst_bnd(env, _, pre, csets, st, SCode.COMPOSITE(crlst), dae)
        => ([DAE.COMPOSITE(el''')], env, csets, st')

end

Figure 5.12.: The inst_boundary relation that instantiates the three different kinds of boundary elements, POINTS, CURVE and COMPOSITE and builds the corresponding DAE structures.
declaration and initialization of parameters, function definitions, creating functions for parameterized boundary description and associating boundary conditions and boundary sections. The generated code is incomplete, and is combined with prewritten C++ code.

Model Parameters

The model parameters declared in functions are collected and generated as fields in separate C++ structs, one for each function instance. The model parameters of the instantiated, complete model are generated as fields in a struct that contains all the parameters of all subcomponents as a flat list of C++ variables. Initialization of all model parameters of all functions is performed in an initialization routine, and another routine initializes the values of the model parameters. The pointer to the model parameter struct is passed to functions so that model parameters can be accessed from within instantiated functions.

Functions

The first step of function code generation is to generate a unique C++ struct for each function, containing the Modelica parameters of that function. A global struct variable is declared at the same time, with a name corresponding to the full name of the function with dots replaced by underscores. Later, initialization code is generated for assigning the parameter values for all function parameters.

Next, the function head is generated. The input variables are generated as arguments of the function with their type and value and output variables are generated as the return value. For simple variable types simple C++ variables are generated, like int, real, etc. For complex types a C++ struct is generated and the name of the struct is used as the type. If there is only one output variable, the type of that variable is used as the return type of the function. If there are more output variables, a return type struct is created and used as the return value containing all the output variables. The return variable is declared at the beginning of the function body and if a struct is used, the output variables are prefixed with the structs name appropriately.

The function body is generated from the algorithm section of the corresponding Modelica function which has a close correspondence to C++ code. Finally, the return statement is constructed.

Function Instances

Code instances of parameterized functions are implemented using the replaceable function feature of Modelica. PDE and boundary condition mod-
5.2. Modelica Translator

els contain replaceable functions that represent coefficients that can depend on space variables. These Modelica functions are instantiated for each instance of a PDE or a boundary condition class, and a unique C++ function is generated in each case. The generated code for each function instance is identical to that for the Modelica function class that the function is an instance of, except that the parameter references are replaced with references to a separate parameter struct.

Domains

A set of C++ classes has been developed in order to simplify code generation for domain classes. The class hierarchy can be seen in Figure 5.13. The three kinds of boundary descriptions use different approaches to code generation, but has the same interface so that the discretization is done automatically. The \texttt{Object2D()} class is the base class containing the interface method \texttt{getPoints()}, which returns a required maximum number of discrete points describing the boundary. The \texttt{Object2D()} class also contains the \texttt{isComposite()} and the \texttt{getCurves()} methods needed for hierarchically composed curves.

![Class hierarchy for the C++ classes written to generate domain descriptions. Line2D objects are represented by the DiscreteCurve2D class.](image)

The \texttt{curve} approach to boundary description uses a parametric expression that defines the boundary curve. The description is symbolic, but is discretized before the triangulation can be made. This is handled by generating a function which calculates the coordinates of a point on the curve for a given parameter value. For example, the boundary description:

\[ \text{curve}(\cos(2\pi u), \sin(2\pi u), u); \]
Chapter 5. Implementation

is translated into a C++ function with the structure shown in Figure 5.14.

```cpp
Point2D func(Real u) {
    Point2D res;
    res.x = cos(2*PI*u);
    res.y = sin(2*PI*u);
    return res;
}
```

Figure 5.14.: Function generated for the boundary described as 
\texttt{curve(cos(2*PI*u), sin(2*PI*u), u)};

The class \texttt{Curve2D} is used to represent parametric curve based descriptions. A new, specialized class is generated for each such domain description, which inherits \texttt{Curve2D} and overloads the \texttt{func()} method to calculate the correct curve. The \texttt{Curve2D} class can be seen in Figure 5.15. The \texttt{getPoints()} method automatically calls the \texttt{func()} method to generate the required number of discrete points on the curve.

```cpp
class Curve2D : public Object2D {
public:
    Curve2D(string name=string("Curve2DObject"))
        : Object2D(name) {
    }

    virtual Point2D func(Real u)=0;
    virtual point_vector* getPoints(int n);
};
```

Figure 5.15.: The \texttt{Curve2D} class that represents parametric curves. Each specific boundary description inherits this class and overloads the \texttt{func()} method.

When the \texttt{lines} method is used for boundary description, the class \texttt{DiscreteCurve2D} is instantiated and the set of points from the \texttt{lines} description is added to that instance. See Figure 5.16 for the contents of the \texttt{DiscreteCurve2D} class. The list of points added to an instance is directly returned from the \texttt{getPoints()} method, ignoring the desired maximum number of points, for simplicity. Hence, it is assumed that the number of points given in the \texttt{lines} operator is always less than the required number of discretization points on the boundary. If this does not hold, the result will be a finer discretization than what is desired.

The \texttt{composite} method is slightly more complex to handle. Figure 5.17 shows the declaration of the \texttt{Composite2D} class that represents composite boundaries. A composite curve is discretized by retrieving its constituent
5.2. Modelica Translator

```c
class DiscreteCurve2D : public Object2D {
    point_vector points;
    public:
    DiscreteCurve2D(string name=string("DiscreteCurve2D"))
        : Object2D(name) {
    }
    void add(const Point2D& p);
    void add(const Point2D* p, int n);
    virtual point_vector* getPoints(int n);
};
```

Figure 5.16.: The DiscreteCurve2D class that represents a set of connected lines. Each specific boundary description instantiates this class and adds the present set of points to that instance using the add() method.

curve segments and discretizing them, compensating for the overlaps at the joints. The getCurves() method is used to access the curve segments. It traverses the composition hierarchy and collects all non-composite curves into a single list. Hence, if a curve segment itself is composite, the getCurves() method is called recursively on that curve object and the resulting list of curves is inserted into the top level list. Thus, the list of curves returned only contains Curve2D or DiscreteCurve2D objects.

```c
class Composite2D : public Object2D {
    curvep_vector curves;
    public:
    Composite2D(string name=string("Composite2D"))
        : Object2D(name) {
    }
    virtual point_vector* getPoints(int n);
    curvep_vector* getCurves();
    void add(Object2D* c);
    void add(Object2D** c, int n);
    virtual bool isComposite() {
        return true;
    }
};
```

Figure 5.17.: The Composite2D class that represents a boundary that is composed of one or more other boundary objects, composite or plain (curve or discrete curve). The getCurves() method recursively goes through any constituent composite curves, returning only a flat list of non-composite curve segments.
Chapter 5. Implementation

Boundary Conditions

Boundary conditions are represented by the BoundaryCondition C++ class, see Figure 5.18. The same class can be used for the three different kinds of boundary conditions: Dirichlet, Robin, and Neumann conditions. The names of the boundary condition objects are mapped from their names in the Modelica domain description. The mesh generator and the PDE solver supports named boundary segments. The code generation consists of instantiating the boundary condition objects and putting them in a list to be passed to the solver.

```cpp
class BoundaryCondition {
    public:
        space_function* g;
        Float q;
        bool is_dirichlet;
        string name;
    public:
        typedef Float space_function (const point& p);
        BoundaryCondition();
};
```

Figure 5.18.: The BoundaryCondition class that is used to represent the three kinds of boundary conditions. The type space_function is used for functions that depend on the space variables. The type point represents the space coordinates in two or three dimensions.

5.3. Numerical Solver

The solution of a PDE problem involves discretization of the domain, mesh generation for the finite element solver, and solution of the discretized problem using a linear equation solver. The mesh generation and the solution is handled by external software, with some adaptation to support the approach taken in this thesis. The class diagram for the different parts of the solution process can be seen in Figure 5.19.

5.3.1. Domain Discretization

Domain discretization is performed by the DiscreteDomain2D class. In the case with a parametric curve description, this involves sampling a given
5.3. Numerical Solver

number of points on the boundary and creating a polygon with these points as the corners of the polygon, in practice approximating the curve with linear segments. The discretization is performed by calling the `getPoints()` method of the `Object2D` interface. The discretizer also keeps track of the boundary section names in order to pass the information to the PDE solver for the boundary conditions to be applied appropriately.

This discretization is done because most mesh generators support the boundary representation based on corner points and linear segments. In the future, splines and other higher-order representations may be supported.

5.3.2. Mesh Generator

The mesh generator interface consists of a function `generate()` which takes an `Object2D` object. The mesh generator classes use the `DiscreteDomain2D` class to generate the polygonal approximation of the boundary from the `Object2D` object. The `DiscreteDomain2D` class traverses the boundary description objects that can be a hierarchy of composite boundary objects and connected lines and curve objects, and generates a flat boundary description of connected lines, i.e. a polygon.

From the discrete domain description, each mesh generator class generates the output in the corresponding format. Two mesh generators are supported, *Bamg* [1], *Easymesh* [3], using the classes *BamgMeshGenerator* and *RheolefMeshGenerator*. The mesh generator class *GeoMeshGenerator* is used to convert a Bamg mesh into the *Geo* format that one of the solvers, Rheolef, uses.
Chapter 5. Implementation

5.3.3. Finite Element Solver

The solver interface contains one function, solve() which takes three arguments: An Object2D object describing the boundary, a list of BoundaryCondition objects and a list of DomainRegion objects. BoundaryCondition objects are named and contain the parameters of the boundary conditions for the boundary sections with matching names. The DomainRegion objects represent different regions in the domain that have different PDE coefficients. Only some of the coefficients can be different in different regions though, depending on which solver that is used.

Three different PDE solvers can be used with this prototype. The different modules involved when using each solver can be seen in Figure 5.20 and Figure 5.21.

Coefficient Based Solver

With the coefficient-based solver, a solver for a predefined, general PDE is used, with the proper parameters from the present PDE problem. Different solvers for different predefined PDEs can be used. In this work Rheolef and FEMLAB are used to solve equations of the form \( \nabla \cdot (c \nabla u) = f \).

Rheolef [39,40] is a publicly available package for solving PDEs using finite element methods. Besides some supporting Unix commands, it contains a set of C++ classes to handle domains, domain boundaries, fields and other PDE related objects. The problem is formulated in variational form by writing a C++ routine that is used together with the Rheolef package.

The PDE problem mentioned above was converted to variational form manually, together with the different possible boundary conditions. The coefficients of the problem are passed to the solver after the translation process. This formulation is independent of the domain geometry, and the occurrence and number of the boundary conditions, as long as the supported kinds of boundary conditions are used. Currently, it handles the three kinds of boundary conditions that are used in the prototype: Dirichlet, Neumann, and Robin conditions. The list of boundary condition objects are passed to the solver which generates the correct problem formulation and further calls the Rheolef equation solver. Rheolef supports meshes generated using Bamg, with a converter that converts the mesh into a Geo mesh, that can be processed by the solver. Boundaries can be referred to by names which is utilized by the PDE Modelica translator. The Rheolef solver routine can be found in Appendix A.

The FEMLAB solver interface is simpler since FEMLAB also uses the coefficient-based problem formulation. The FEMLAB programming interface is used and coefficient assignments are generated into a Matlab script
5.3. Numerical Solver

Mesh generation is handled internally by FEMLAB, but the discrete domain description is transferred through the same interface and a FEMLAB polygon object is generated. A special routine was written though, in order to rearrange the boundary condition indices on the edges because FEMLAB’s polygon creation routine sorts the boundary vertices which invalidates the boundary condition indices on the edges. The rearrange routine finds the vertices according to their coordinates and determines the new edge indices so that the boundary conditions are applied correctly. This routine is available in Appendix B.

Finally, mesh generation and solution commands are generated and put into the Matlab script file. The script can then be executed and FEMLAB’s post-processing tools can be used to analyze the solution. An example of a generated file can be seen in Appendix C.

Solver Generator in Mathematica

The Mathematica-based solver generator supports general Mathematica differential equation formulation of a PDE. Hence, no manual conversion to variational form is necessary, and the solver can be used for PDEs with different structure. Mathematica is used for symbolic processing of the equations and generating a finite element solver for the specific domain. The resulting equation system can be solved by Mathematica’s built-in linear solver or an external solver such as SuperLU.

Both a finite difference method [41,42] implementation and a finite element method implementation exist, as well as a method of lines implementation using finite differences. The finite difference and the method of lines solvers are used with a translator implemented in MathModelica [18,25].

The current implementation of the translator uses the finite element implementation, MathFEM, but complete equation generation is not yet supported. Instead, a coefficient based PDE is generated together with the coefficient value settings. The problem description generation is similar to that of FEMLAB described previously, but here a Mathematica script file is generated. Appendix D contains a listing of a generated example.

There is also ongoing work on a solver in Mathematica based on the method of lines using finite elements. Here, space discretization is performed by the solver which generates only time dependent ordinary differential equations. Standard Modelica code can be generated from this solver that can be solved further using existing Modelica environments, for example the modeling and simulation tool Dymola [2], the Open Source Modelica environment [5,16,31], or the MathModelica environment [18].
Chapter 5. Implementation

Figure 5.20.: Solution method using an external PDE solver or a pre-generated solver. This method is used with Rheolef and FEM-LAB.
5.3. Numerical Solver

Figure 5.21.: Solution method with dynamic solver generation. This method is used with the MathFEM solver.
5.4. Future Extensions in the Implementation

Some parts of the prototype implementation are not completed. The mixed ODE and PDE based models cannot be translated automatically yet. The missing part is a method of lines solver implementation to convert the PDE models into ODE models and merge with the rest of the ODE models, maintaining the connections between the models.

Representation of complex domains is not yet implemented in PDEModelica and the translator, and incomplete in the mesh generator interface. Although support for complex domains with subdomains is included in the PDE solver interface.

Another issue is that the \texttt{composite()} operator has the following inconsistent properties:

- defining a boundary segment consisting of other boundary segments.
- increasing the dimension of the domain, i.e. the domain containing the \texttt{composite()} operator automatically becomes a two-dimensional domain. To assure a correct domain, open composite boundaries are automatically closed, using a line segment between the end points.

Thus, complex boundary segments that are not treated as one-dimensional boundary domains, and which can be used as boundaries in other domains, are not supported. The correct semantics of \texttt{composite()} yet to be implemented is to define a possibly non-closed boundary segment consisting of several parts.
Chapter 6.

Examples

This chapter contains three examples solved using the prototype implementation. The static current example in Section 6.2 was only partially defined in PDEModelica, because of lack of subdomain support. The heat transfer example in Section 6.3 was solved using both Rheolef and FEMLAB for comparison. Time-dependent problems can be solved using FEMLAB, and an experimental time-dependent solver is implemented for Rheolef, but no time-dependent examples are mentioned here.

6.1. Electrostatic Fields

Given the electric potential on the boundary of a geometric domain, the electrostatic field inside the domain can be calculated. The relationship between the conservative electric field intensity \( \mathbf{E} \), which is a vector field, and the electric scalar potential \( V \) is as follows:

\[
\mathbf{E} = -\nabla V
\]  

(6.1)

The simplification of Maxwell’s equations for electrostatic fields gives the equations

\[
\nabla \cdot \mathbf{D} = \rho \\
\mathbf{D} = \varepsilon \mathbf{E}
\]

where \( \mathbf{D} \) is the electric flux density, \( \rho \) the charge density, and \( \varepsilon \) the electric permittivity of the specific material where the electric field is studied. Using equation (6.1) with these gives

\[
-\nabla \cdot (\varepsilon \nabla V) = \rho
\]

This is Poisson’s equation defining the electric potential distribution in a material with electric permittivity \( \varepsilon \) and the charge density \( \rho \). If the charge density is zero, the Laplace equation is obtained.
Consider a high voltage substation in the vicinity of an electrically grounded transformer [20], illustrated in Figure 6.1. The line labeled A represents the high voltage part with the potential \( V = 500 \text{kV} \), and line B defines the transformer and the ground with \( V = 0 \). The lines C and D are chosen to be far away from the transformer, where the electrical field becomes approximately vertical and thus tangential to the lines C and D. No change in electric field intensity occurs across lines C and D because of the Neumann boundary conditions. The lines A and B have constant potential, corresponding to Dirichlet boundary conditions.

The PDEModelica code for this example is shown in Figure 6.2. The model is translated, compiled, and executed. The resulting plot can be seen in Figure 6.3.

### 6.2. Static Currents

This example illustrates calculation of current distribution in a conductor consisting of two layers with different materials, copper and aluminum. When the current difference between two boundaries is known, the distribution can be calculated by the following equation in the two-dimensional case, involving the electrical vector potential \( \mathbf{T} \):

\[
\frac{\partial}{\partial x} \left( \frac{1}{\sigma} \frac{\partial \mathbf{T}}{\partial x} \right) + \frac{\partial}{\partial y} \left( \frac{1}{\sigma} \frac{\partial \mathbf{T}}{\partial y} \right) = 0 \tag{6.2}
\]

\[
\n\]
6.2. Static Currents

domain VoltageLine
extends Domain2D;
boundary
line({{4.5, 2.}, {2., 2.}, {2., 1.0}, {-0.5, 1.0}, {-1.5, 2.},
{-4.5, 2.}});
end VoltageLine;
domain Transformer
extends Domain2D;
boundary
line({{-4.5, -2.}, {-1., -2.}, {-1., -1.5}, {-1.5, -1.}, {-1.5, 0.},
{1.5, 0.}, {1.5, -1.}, {1., -1.5}, {1., -2.}, {4.5, -2.}});
end Transformer;
domain TransformerDomain
extends Domain2D;
VoltageLine voltageline;
Transformer transformer;
Line2D left (x0=-4.5, y0= 2, x1=-4.5, y1=-2);
Line2D right(x0= 4.5, y0=-2, x1= 4.5, y1= 2);
boundary
composite(voltageline, left, transformer, right);
end TransformerDomain;

model ElectricField
parameter Real rho=0, epsilon=1;
extends PDECoeff2D(c=epsilon, f=rho);
end ElectricField;

model PDEModel
Dirichlet voltageline(g=500), ground(g=0);
Neumann verticalfield;
ElectricField ef;
equation
dom.eq=ef;
dom.left.bc=verticalfield;
dom.right.bc=verticalfield;
dom.voltageline.bc=voltageline;
dom.transformer.bc=ground;
end PDEModel;

Figure 6.2.: PDEModelica code for the electric field calculation example in Section 6.1 with a high voltage substation and a grounded transformer.
Chapter 6. Examples

Figure 6.3.: Solution of the transformer problem defined in Section 6.1.

The domain geometry can be seen in Figure 6.4. The current is known on the boundaries C and D where Dirichlet boundary conditions are applied. On A and B where \( \mathbf{E} \) is perpendicular to the boundary, Neumann boundary conditions are applied. The solution with the current lines can be seen in Figure 6.5. The result shows that the current in the copper layer is approximately 60% higher than in the aluminum layer, which is a result of the conductivity of copper being approximately 60% higher than the conductivity of aluminum (\( \sigma_{\text{copper}} = 5.7 \times 10^7 (\Omega m)^{-1} \) and \( \sigma_{\text{aluminum}} = 3.5 \times 10^7 (\Omega m)^{-1} \)).

This example is only partially implemented in PDEModelica, because of lack of support for complex domains, see Section 4.3.2 and Section 7.2.1. The missing parts of the domain geometry description was manually inserted into the generated intermediate files.

6.3. Heat Transfer

An example similar to the heat transfer example described in Section 1.1 is solved with a more complex boundary for illustration. The outside temperature of 20\(^\circ\)C is used and the constant temperature of 60\(^\circ\)C is used on the middle section of the bottom boundary. The right boundary is the non-insulated side, the other sides are perfectly insulated. The PDEModelica code for the boundary description is shown in Figure 6.8. This problem was solved with two different solvers for comparison, the solution using Rheolef can be seen in Figure 6.6 and the solution from FEMLAB is shown in Figure 6.7. The generated FEMLAB input file for this problem is listed in Appendix C.
6.3. Heat Transfer

Figure 6.4.: A static current problem. The current on the boundaries C and D is given, and the current distribution inside the domain is calculated.

Figure 6.5.: Solution of the static current problem defined in Section 6.2.
Chapter 6. Examples

Figure 6.6.: Solution of the stationary heat conduction example in Section 6.3 using Rheolef.

Figure 6.7.: Solution of the stationary heat conduction example in Section 6.3 using FEMLAB.
6.3. Heat Transfer

Figure 6.8.: The domain geometry for the example in Section 6.3. See Figure 4.3 and Figure 4.2 for the definitions of Bezier2D and Line2D, respectively.
Chapter 6. Examples
Chapter 7.

Conclusions and Future Work

This chapter presents some conclusions drawn from the results of this research, and outlines future work that is needed to achieve a more complete solution to the problem.

7.1. Conclusions

The language extensions described in this thesis and implemented in the prototype suffice to solve PDE problems based on a general PDE with coefficients that can be modified. The three most common types of boundary conditions are supported, which allows formulation and solution of a wide range of problems that are based on the predefined PDE. Complex domain boundaries are supported thanks to the curve() operator which allows a parametric description of the boundary, and the composite() operator which can combine boundary segments of curves and lines.

7.2. Future Work

Possible future research directions are discussed in this section.

7.2.1. Complex Domains

Support for complex domains as discussed in Section 4.3.2 will be added to the language and implemented in the prototype. There is currently some basic support in the solver part of the prototype for subdomains with different PDE parameters, i.e. the use of domain regions, but the language definition and the translator must be modified to support subdomains. This issue is somewhat related to the issue of generalized, spatially distributed connectors. Subdomains and separate domains that are connected are two
approaches to describe problems where different PDEs, or the same PDEs with different coefficients, occur in different subdomains.

7.2.2. Generalized Connectors

The current connector concept in Modelica can be generalized to support connectors with spatial distribution. The static current calculation in Section 6.2 is an example where a connection would be used between two kinds of physical materials. Such a connector needs to be defined with respect to two issues: the geometric description of the interface and the mathematical behaviour of the model across the interface are required. Modelica connectors are also acausal, i.e. the unknown variable is not predetermined but defined by how the components are connected in the final model. This feature should be maintained for generalized connectors.

Additional solver support is also needed to test the extensions in the prototype. One possible approach for solving such models can be to use the interface relaxation method [34].

A connection mechanism to connect ODE-based models and PDE-based models is a very important feature that is needed in order to integrate PDE-based models into existing Modelica models and to simulate combined ODE- and PDE-based models, for example ODE based control systems. The example shown in Section 1.1.1 illustrates a combined ODE and PDE based system. The solution of mixed ODE and PDE models can be implemented by a method of lines solver that converts the PDEs into ODEs, which can be merged with the rest of the ODE models for solution with existing Modelica implementations.

7.2.3. Expressing PDEs

The operators presented in Section 4.4.1 needed for expressing PDEs and boundary conditions explicitly in the language will be implemented and evaluated. This will allow explicitly written PDEs in models, which is different from the current, coefficient-based approach with only parameter modification. Furthermore, the PDE solver interface must be extended to support transfer of equations to the solver.

7.2.4. Debugging

There is ongoing work on debugging of declarative equation based languages, specifically debugging of Modelica models [6,7]. It is useful to also investigate debugging of PDE based models, for example checking of sufficient definition of initial and boundary conditions.
References


References


References


References


Appendix A.

PDE Solver Using Rheolef

Rheolef specific parts from RheolefSolver.cc:

```cpp
void makeXh(geo & omega,
            const string& eltype,
            const vector<BoundaryCondition> & bclist,
            space & Xh) {

domain toblock;
int ndir=0;

for (unsigned int i=0; i<bclist.size(); i++) {
    domain d = omega[bclist[i].getName()];
    if (bclist[i].is_dirichlet) {
        ndir++;
        toblock += d;
    }
}
if (ndir > 0)
    Xh.block(toblock);
}
```

/Steady-state solver. eltype is one of the element types used in Rheolef, e.g. "P1". Each DomainRegion contains the "c" and "f" coefficients of the PDE. Only different "c" coefficients are supported (same "f" for the entire domain).

```cpp
void heatgeneric(geo & omega,
                 const vector<BoundaryCondition> & bclist,
                 const string& eltype,
                 DomainRegion::space_function_ptr f,
                 const vector<DomainRegion> & reglist) {
```
Appendix A. PDE Solver Using Rheolef

```cpp
int nneumann=0, nrobin=0;

space Xh(omega, eltype);
makeXh(omega, eltype, bclist, Xh);

space Qh (omega, "P0", "vector");
field eta (Qh, 1);

if (reglist.size() == 1) {
    eta = reglist[0].pde>c;
} else {
    for (unsigned int i=0; i<reglist.size(); i++) {
        cout << reglist[i].name << " = " << reglist[i].pde>c;
    }
}

form diag d (eta);
form grad (Xh, Qh, "grad");
form m (Xh, Xh, "mass");
form inv_m (Qh, Qh, "inv_mass");

field fh = interpolate (Xh, f);
field uh (Xh);
vec<Float> solvec = m.uu*fh.u + m.ub*fh.b;

form a = -trans(grad)*inv_m*grad;

for (unsigned int i=0; i<bclist.size(); i++) {
    domain d = omega[bclist[i].getName()];
    space Wh (omega, d, eltype);
    field gh = interpolate(Wh, bclist[i].g);
    if (bclist[i].is_dirichlet) {
        uh[d] = gh;
    } else {
        form mb (Wh, Xh, "mass_bdr");
        solvec += mb.uu*gh.u + mb.ub*gh.b;
        if (bclist[i].q == 0)
            nneumann++;
        else {
            nrobin++;
            form ab (Xh, Xh, "mass_bdr", d);
            a = a + bclist[i].q*ab;
        }
    }
}
```

84
solvec = a.ub + uh.b;

ssk<Float> fact = ldlt(a.uu);
uh.u = fact.solve(solvec);

cout << uh;
}

/*
 * Time dependent solver. Time derivative is approximated
 * with the backward Euler scheme (Langtangen p.123).
 * Solution from the previous time step must be sent in the
 * uprev parameter.
 */
void heatgenericetime(geo & omega,
    const vector<BoundaryCondition> & bclist,
    const string & eltype,
    Float da,
    Float c,
    BoundaryCondition::space_function_ptr f,
    Float dt,
    field & uprev,
    const vector<DomainRegion> & reglist ) {

    int neumann=0, nrobin=0;

dt = dt / da;

    space Xh(omega, eltype);
    makeXh(omega, eltype, bclist, Xh);

    space Qh (omega, "P0", "vector");
    field eta (Qh);

    if (reglist.size() == 1) {
        eta = reglist[0].pde->c;
    }
    else {
        for (unsigned int i=0; i<reglist.size(); i++) {
            eta[reglist[i].name] = reglist[i].pde->c;
            cout << reglist[i].name << "u" << reglist[i].pde->c;
            cout << endl;
        }
    }

    form diag d (eta);

    form grad (Xh, Qh, "grad");
    form m (Xh, Xh, "mass");
    form invm (Qh, Qh, "invmass");

85
Appendix A. PDE Solver Using Rheolef

```cpp
field fh = dt * interpolate (Xh, f) + uprev;
field uh (Xh);
vec<Float> solvec = m.uu + m.ub + fh.b;
form ut (Xh, Xh, "mass");
form a = ut - dt * trans (grad) * (inv m * d) * grad;
// theta = 1, backward euler scheme. See Langtangen p.123
for (unsigned int i=0; i<bclist.size(); i++)
    domain d = omega[bclist[i].getName()];
space Wh (omega, d, eltype);
field gh = interpolate (Wh, bclist[i].g);
if (bclist[i].isDirichlet)
    uh[d] = gh;
else {
    form mb (Wh, Xh, "mass_bdr");
    mb = dt * mb;
    solvec += mb.u + mb.ub + gh.b;
    if (bclist[i].q == 0)
        neumann++;
    else {
        nrobin++;
        form ab (Xh, Xh, "mass_bdr", d);
        a = a + bclist[i].q * dt * ab;
    }
}
ssk<Float> fact = ldlt (a.uu);
uh.u = fact.solve (solvec);
uprev = uh;
```
Appendix B.

FEMLAB Domain Generation Interface Routine

The Matlab routine `femlabdomainfix` for reordering of the boundary conditions after FEMLAB’s vertex sorting. The input variables are the array of points with coordinates and boundary condition indices \( v \), the array of edges with vertex indices and boundary condition indices and \( e \). The output variables are the generated FEMLAB polygon object \( poly \) and the array \( neweb \) of boundary condition indices with the correct order according to the created polygon.

```matlab
femlabdomainfix.m:

vx = v(:,1);
vx = v(:,2);
vb = v(:,3);
es = e(:,1);
es = e(:,2);
es = e(:,3);

poly = line2(vx,vx);
newvertices = fgeomvtx(poly)';
newedges = fgeomse(poly)';

newvx = newvertices(:,1);
newvy = newvertices(:,2);
newes = newedges(:,1);
newee = newedges(:,2);

n = length(newes);
neweb = ones(n,1);
```
Appendix B. FEMLAB Domain Generation Interface Routine

\[ \text{myeps} = \frac{\varepsilon}{100}; \]

\[
\text{for } i = 1:n \\
\hspace{1cm} \text{news} = \text{newes}(i); \\
\hspace{1cm} \text{newe} = \text{newee}(i); \\
\hspace{1cm} \text{newsx} = \text{newvx}(\text{news}); \\
\hspace{1cm} \text{newsy} = \text{newvy}(\text{news}); \\
\hspace{1cm} \text{newex} = \text{newvx}(\text{newe}); \\
\hspace{1cm} \text{newey} = \text{newvy}(\text{newe}); \\
\hspace{1cm} \text{for } j = 1:n \\
\hspace{2cm} \text{s2} = \text{es}(j); \\
\hspace{2cm} \text{e2} = \text{ee}(j); \\
\hspace{2cm} \text{sx} = \text{vx}(\text{s2}); \\
\hspace{2cm} \text{sy} = \text{vy}(\text{s2}); \\
\hspace{2cm} \text{ex} = \text{vx}(\text{e2}); \\
\hspace{2cm} \text{ey} = \text{vy}(\text{e2}); \\
\hspace{1cm} \text{if } \left( (\text{abs(\text{newsx} - \text{sx}) < \text{myeps}) \& (\text{abs(\text{newsy} - \text{sy}) < \text{myeps})} \& \ldots \right) \\
\hspace{1cm} \left( (\text{abs(\text{newex} - \text{ex}) < \text{myeps}) \& (\text{abs(\text{newey} - \text{ey}) < \text{myeps})} \& \ldots \right) \\
\hspace{1cm} \left( (\text{abs(\text{newsx} - \text{sx}) < \text{myeps}) \& (\text{abs(\text{newsy} - \text{sy}) < \text{myeps})} \& \ldots \right) \\
\hspace{2cm} \text{neweb}(i) = \text{eb}(j); \\
\hspace{1cm} \text{end} \\
\hspace{1cm} \text{end} \\
\hspace{1cm} \text{end} \]
Appendix C.

Problem Formulation Generated for FEMLAB

The generated problem formulation code for the heat transfer example in Section 6.3 for solving with FEMLAB. The arrays v and e are passed to femlabdomainfix.m, shown in Appendix B, which creates the FEMLAB polygon in poly and puts the boundary conditions in neweb.

femlabproblem.m:

```matlab
clear fem;
fem.dim = {'u'};
fem.form = ' coefficient ';
fem.equ.f = 0;
fem.equ.c = 1;
bnd.g = {{{'1.08e+06'}}, {{'0'}}, {{'0'}}, {{'0'}}};
bnd.q = {{{'30000'}}, {{'0'}}, {{'0'}}, {{'0'}}};
bnd.h = {{{'0'}}, {{'0'}}, {{'0'}}, {{'0'}}};
bnd.r = {{{'0'}}, {{'0'}}, {{'0'}}, {{'0'}}};
```

\(v = [1 -2 1.098529 -1.70911 1.087162 -1.50575 1.0870313 -1.36317 1.0407803 -1.25928 1.0104429 -1.17629 1.0221312 -1.10036 1.0552779 -1.02119 1.0875175 -0.93169 1.0875175 -0.827545 1.044283 -0.706894 1.0166774 -0.569928 1.0418526 -0.196275 -0.255877 1.020363 1.0.861045 1.02036 0.0861045 1.0196275 0.255877 1.018429 0.418526 1.0166774 0.569928 1.044283 0.706894 1.01755 0.827545 1.0.875175 0.93169 1.0.552779 1.02119 1.0.221312 1.10036 1.0.104429 1.17629 1.0.407803 1.25928 1.0.670313 1.36317 1.0.87162 1.50575 1.0.989529 1.70911 1.0870313 -1.128989]\

89
Appendix C. Problem Formulation Generated for FEMLAB

2; -1.21656 1.99413 3; -1.43248 1.97655 3; -1.64713 1.94731 3; -1.85988 1.90648 3; -2.07011 1.8542 3; -2.27721 1.79061 3; -2.48055 1.71591 3; -2.67956 1.6303 3; -2.87363 1.53405 3; -3.06222 1.42743 3; -3.24475 1.31076 3; -3.4207 1.18437 3; -3.58955 1.04865 3; -3.7508 0.903982 3; -3.90398 0.750798 3; -4.04865 0.589545 3; -4.18437 0.420697 3; -4.31076 0.244748 3; -4.42743 0.0622154 3; -0.320444 3; -4.71591 -0.519447 3; -4.79061 -0.722794 3; -4.8542 -0.929887 3; -4.90648 -1.14012 3; -4.94731 -1.35287 3; -4.97655 -1.56752 3; -4.99413 -1.78344 3; -5 -2 3; -3 -2 4; -1 -2 5];
e = [ 1 2 1; 2 3 1; 3 4 1; 4 5 1; 5 6 1; 6 7 1; 7 8 1; 8 9 1; 9 10 1; 10 1; 11 1; 11 12 1; 12 13 1; 13 14 1; 14 15 1; 15 16 1; 16 17 1; 17 18 1; 18 19 1; 19 20 1; 20 21 1; 21 22 1; 22 23 1; 23 24 1; 24 25 1; 25 26 1; 26 27 1; 27 28 1; 28 29 1; 29 30 1; 30 31 1; 31 32 1; 32 33 1; 33 34 1; 34 35 1; 35 36 1; 36 37 1; 37 38 1; 38 39 1; 39 40 1; 40 41 1; 41 42 1; 42 43 1; 43 44 1; 44 45 1; 45 46 1; 46 47 1; 47 48 1; 48 49 1; 49 50 1; 50 51 1; 51 52 1; 52 53 1; 53 54 1; 54 55 1; 55 56 1; 56 57 1; 57 58 1; 58 59 1; 59 60 1; 60 61 1; 61 62 1; 62 1 6 ];
myhmax = 0.1;
run ../../../solver/femlabdomainfix.m;
bnd.ind = neweb;
fem.bnd = bnd;
fem.geom = poly;
fem.mesh = meshinit(fem, 'hmax', myhmax);
fem.sol = femlin(fem);
Appendix D.

PDE Formulation Generated for Mathematica Based Solver

The generated problem formulation code for the heat transfer example in Section 6.3 for the Mathematica based FEM solver. Two files are generated, the first one for the solver generation and then code generation using MathCode, and the second one for the execution of the solver.

Solver generation code:

```
Needs["MathFEM"];
parameters = { }; partypes = { };
pdeeq = { 1/6 (D[v[x, y], {x, 2}] + D[v[x, y], {y, 2}]) == 0};
boundaryconditions = {
{1, 1/6 OutwardNormalDer[v[x, y], 1] + 30000 * v[x+y] == 1.08 e+06},
{2, 1/6 OutwardNormalDer[v[x, y], 1] == 0},
{3, 1/6 OutwardNormalDer[v[x, y], 1] == 0},
{4, 1/6 OutwardNormalDer[v[x, y], 1] == 0},
{5, v[x, y] == 60},
{6, 1/6 OutwardNormalDer[v[x, y], 1] == 0}
} ;
equations = Prepend[boundaryconditions, pdeeq];
meshdata = EasyMeshData["mathfemmesh.n", "mathfemmesh.e", "mathfemmesh.s"], 6];
{Nn, Ne, nmark, maxsize1, maxsize2, matsize, InteriorSize} = {
Length[meshdata[[5]]],
Length[meshdata[[6]]],
Length[meshdata[[1]]],
Apply[Max, Map[Length, meshdata[[1]]]],
Apply[Max, Map[Length, meshdata[[1]]]],
Apply[Plus, Append[meshdata[[3]], meshdata[[4]]]],
meshdata[[4]]}
```
Appendix D. PDE Formulation Generated for Mathematica Based Solver

\[\text{Attributes[declare]} = \{\text{HoldAll}\};\]
\[
\text{statement = Statement[declare, partypes, Nn, Ne, nmark, maxsize1, maxsize2, matsize, InteriorSize];}\]
\[
\text{Clear[mat1];}\]
\[
\text{mat1}[n1_, n2_] := \text{MATRIX}[\{1 | n1, 1 | n2\}];}\]
\[
\text{Needs["MathCode"]};\]
\[
(\text{BeginPackage["MathFEM","\{ToExpression["MathCodeContexts"]\}]};}\]
\[
\text{Begin["MathFEM"]};\]
\[
\text{statement /. declare : ToExpression["Declare"]};}\]
\[
\text{End[]};\]
\[
\text{Begin["MathFEM'Private"]};\]
\[
\text{End[]}];\]
\[
\text{EndPackage[]}];\]
\[
\text{BuildCode["MathFEM", Language -> "C++"]};\]

Solver execution code:

\[
\text{Needs["MathFEM"]};\]
\[
\text{Off[Syntax::"sntoct1"]};\]
\[
\text{Needs["MathCode"]};\]
\[
\text{InstallCode["MathFEM"]};\]
\[
\text{meshdata = EasyMeshData[\{"mathfemmesh.n","mathfemmesh.e","mathfemmesh.s"\}, 6]};\]
\[
\{\text{Nn, Ne, nmark, maxsize1, maxsize2, matsize, InteriorSize}\} = \{
\text{Length[meshdata[[5]]]},
\text{Length[meshdata[[6]]]},
\text{Length[meshdata[[1]]]},
\text{Apply[Max, Map[Length, meshdata[[1]]]}],
\text{Apply[Max, Map[Length, meshdata[[7]]]}],
\text{Apply[Plus, Append[meshdata[[3]], meshdata[[4]]]}],
\text{meshdata[[4]]}\};\]
\[
\text{paramvalues = \{\};}\]
\[
\text{Apply[GenerateMatrices, Join[meshdata, paramvalues]]};\]
\[
\text{lhs = Table[list1[i, j], \{i, 1, matsize\}, \{j, 1, matsize\}]};\]
\[
\text{rhs = Table[list1[i, matsize+1], \{i, 1, matsize\}]};\]
\[
\text{solution = LinearSolve[lhs, rhs]}];\]
\[
\text{solution >> "solution.m"};\]
PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations

Levon Saldamli

This thesis describes initial language extensions to the Modelica language to define a more general language called PDEModelica, with built-in support for modeling with partial differential equations (PDEs). Modelica® is a standardized modeling language for object-oriented, equation-based modeling. It also supports component-based modeling where existing components with modified parameters can be combined into new models. The aim of the language presented in this thesis is to maintain the advantages of Modelica and also add partial differential equation support.

Partial differential equations can be defined using a coefficient-based approach, where a predefined PDE is modified by changing its coefficient values. Language operators to directly express PDEs in the language are also discussed. Furthermore, domain geometry description is handled and language extensions to describe geometries are presented. Boundary conditions, required for a complete PDE problem definition, are also handled.

A prototype implementation is described as well. The prototype includes a translator written in the relational meta-language, RML, and interfaces to external software such as mesh generators and PDE solvers, which are needed to solve PDE problems. Finally, a few examples modeled with PDEModelica and solved using the prototype are presented.
Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
Dan Fristedt: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
Staffan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
Kristina Larsen: Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.
Niclas Andersson: Compilation of Mathematical Models to Parallel Code, 1996.
Jörgen Lindström: Chefers användning av kommunikationsteknik, 1996.
Niclas Wahlfö: A Default Extension to Description Logics and its Applications, 1996.
FiF-a 4  Carita Åbom: Videomötesteknik i olika affärsorganisationer - möjligheter och hinder, 1997.
No 627  Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
No 653  Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
No 668  Per-Ove Zetterlund: Normaling av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
FiF-a 16  Marie-Therese Christiansson: Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
FiF-a 21  Ulf Seigerroth: Integration of reconfiguration methods - a model for well-founded methodintegration, 1999.
No 742  Pawel Pietrzak: Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748  Declarative Error Diagnosis of GAPLog Programs, 1997.
No 787  Charlotte Björkegren: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 807  Svein Bergum: Managerial communication in telework, 2000.
No 820  Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.
No 842  Magnus Kald: The role of management control systems in strategic business units, 2000.
FiF-a-49  Stefan Holgersson: IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51  Per Oscarsson: Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 931  Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 938  Bourhane Kadmiry: Fuzzy Control of Unmanned Helicopter, 2002.
No 964  Peter Bunus: Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.