L. Saldamli, P. Fritzson:
Object-oriented Modeling with Partial Differential Equations.
Modelica Workshop 2000 Proceedings, pp. 119-128.

Paper presented at the Modelica Workshop 2000, Oct. 23.-24., 2000, Lund, Sweden.

All papers of this workshop can be downloaded from
http://www.Modelica.org/modelica2000/proceedings.html

Object-oriented Modeling with Partial Differential Equations

Levon Saldamli and Peter Fritzson
PELAB, Programming Environments Laboratory
Department of Computer and Information Science
Linköping University
SE-581 83, Linköping, Sweden.
levsa@ida.liu.se, petfr@ida.liu.se

Abstract

Mathematical models containing partial differential equations (PDEs) occur in many engineering applications. Modelica is a general, high-level language for object-oriented modeling with differential-algebraic equations (DAEs). There is a need for extending Modelica to also support modeling with PDEs. This paper presents some ideas on such extensions to the Modelica language, that would allow formulation of PDE problems defined on general domains using objects.

1 Introduction

Modelica [4, 5, 3, 1] is a general, high-level language for object-oriented modeling with differential-algebraic equations (DAEs). The aim of the Modelica language is to represent models in a general, domain-independent way. Using object-oriented constructs, Modelica supports reuse of existing models and model libraries. By introducing support for PDEs in Modelica, these features will be useful also for modeling and simulation of PDE-based models.

There are several low-level PDE solving packages in which the problem is specified as program code written in a programming language (e.g. Diffpack, Overture), as well as high-level packages where special purpose languages are used for formulation of the problem (e.g., gPROMS [10], ELLPACK [12], PDEQSOL [6]), and interactive tools such as FEMLAB.

This section gives a brief introduction to PDEs and numerical solution methods, and mentions other existing packages with built-in high-level PDE language support. The MathModelica environment that is used for implementation is also introduced in this section. Section 2 contains a discussion of proposed extensions to Modelica. In Section 3, current status of our work is given, and in Section 4 future plans are presented.

1.1 Partial Differential Equations (PDEs)

An initial and boundary value problem consists of a partial differential equation and a set of initial and boundary conditions. A simple two-dimensional example using the wave equation is

$$\frac{\partial^2 U}{\partial t^2} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}, \qquad (x,y) \in \Omega$$

where Ω is the domain, with the boundary condition

$$u(x,y,t) = 0, \qquad (x,y) \in \partial\Omega$$

where $\partial\Omega$ is the boundary of the domain, and the initial condition

$$u(x,y,0) = f(x,y), \qquad (x,y) \in \Omega$$

The first derivative with respect to the variable x is compactly written as $\partial_x U$, and the second derivative as $\partial_{xx} U$, or even shorter as U_x and U_{xx}.

1.2 Numerical Solution of PDEs

With existing theory, only a small number of PDEs can be solved analytically, and even smaller number of these is solvable with useful boundary conditions. Therefore, many methods have been developed for solution of numerical approximations of PDEs.

1.2.1 Finite Difference Methods

In order to find the unknown function U satisfying the PDE, the domain is discretized using a grid of points, searching the value of U at each grid point. In a two-dimensional domain with NxN grid points, the value of U at each point is

$$u_{i,j} = U(x_i, y_j), \qquad i = 0,\ldots,N-1, j = 0,\ldots,N-1$$

Using Taylor's theorem, derivatives can be approximated by difference quotients. The first derivative of

U with respect to x is then, using a *central-difference* formula,

$$\partial_x U \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}$$

Other approximations can be obtained with the *forward-difference* formula

$$\partial_x U \approx \frac{u_{i+1,j} - u_{i,j}}{\Delta x}$$

and the *backward-difference* formula

$$\partial_x U \approx \frac{u_{i,j} - u_{i-1,j}}{\Delta x}$$

An *explicit* solver iterates over the points, calculating $u_{i,j}$ at each point using previously calculated values. An *implicit* solver, when iterating over the points, needs to solve a system of equations for a set of points in each iteration, in order to calculate the values of u. Iterating solvers have different convergence properties, depending on the specific PDE and the selected difference method.

1.2.2 Finite Element Methods

One way of solving a PDE numerically is to approximate the unknown U with a sum of basis functions, for example piece-wise polynomial functions, with unknown factors:

$$\hat{U}(x) = \sum_{k=0}^{N} a_k \phi(x)$$

A system of equations is then set up with the factors a_k as unknowns. Solving this system gives the approximate solution $\hat{U}$. The family of methods based on this method is called *weighted residual methods*, and different choices of weighting functions gives different methods, such as the *collocation method*, the *least-squares method* and the *Galerkin method*.

Instead of finding a global solution, the domain can be divided into smaller domains, called *elements*, with additional conditions on the new boundaries, which can be solved separately using the same methods as in the global case. This method is called the *finite element method*. Subdivision of the domain can be done in different ways, e.g. by triangulating the domain. Sizes of the elements can also be adjusted according to expected variations of U in different parts of the domain, creating smaller elements where fast variations are expected. Thus, few polynomials with low order are often sufficient to approximate U over each element.

1.2.3 Method of Lines

Time dependent PDEs can be discretized with respect to the space variables, generating a set of ODEs, which can be solved using ODE solvers. This method is called the *method of lines*. The advantage of this method is that there exists advanced algorithms for fast and accurate solution of ODEs, e.g. with automatic step adjustment to match a requested error tolerance. However, a drawback is that these solvers do not have control over the error introduced by the initial discretization that generates the ODEs, which can be much bigger than the error of the solution of ODEs.

1.3 Related Work

An early effort to define a language for partial differential equations was PDEL [2] in 1970. Since then, a number of approaches have been investigated to create high-level language based environments to simplify specification and solution of PDE-based problems. In this section, we summarize some of these environments.

1.3.1 ELLPACK

ELLPACK [12] is a package for solving elliptic partial differential equations. It defines a high-level language for specification of a PDE, a boundary, a grid and a discretization method as well as a solution method. In this language, there are predefined names for the solution variable and its derivatives, such as `U`, `UX` ($\partial_x U$), `UYY` ($\partial_{yy} U$) and the spatial variables `X`, `Y` and `Z`. These are used to specify the PDE:

```
EQUATION.   UXX + UYY + 3.0*UX - 4.0*U = &
               EXP(X+Y)*SIN(PI*X)
```

Here '&' is used as a line continuation marker for long entries. The boundary conditions can be specified in different ways, using lines and parametric curves. A circular domain is specified as:

```
BOUNDARY. U=0.0 ON X= 1.-COS(PI*THETA), &
                  Y= 1.-SIN(PI*THETA) &
                  FOR THETA = 0. TO 2.
```

This boundary specifies the condition that the function is zero on the circle with the radius 1 and the center $(1,1)$. The `BOUNDARY` statement also defines the domain of the problem. The `EQUATION` and `BOUNDARY` parts of an ELLPACK program is declarative, whereas the rest (`GRID`, `DISCRETIZATION` etc.), are executed in a sequential order.

Furthermore, the *PELLPACK* [7] (Parallel ELLPACK) problem solving environment has been developed, also based on ELLPACK, supporting domain-decomposition based parallel solvers and finite element methods. It includes several PDE solving systems, some of those with several specific solving methods. It also covers parabolic problems besides elliptic ones, and some hyperbolic problems. As a problem solving environment (PSE), it also has support for graphical user interfaces, visualization and analytic tools.

1.3.2 PDESpec

S. Weerawarana [14] presents a high-level language *PDESpec* where the PDE problem is specified using objects. Different types of objects are *equation, domain, boundary* and *initial conditions, mesh, decomposition, algorithm, solve* and *solution.* For example, an equation object describing the steady-state heat flow is defined as:

```
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] ]
 );
```

Here Tx represents $\partial_x T$ and `Dx(A)` is an alias for `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 domain object is specified as:

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

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

1.3.3 gPROMS

The *gPROMS* environment is used for dynamic simulation of chemical processes, and a modeling language was designed and implemented by M .Oh [10] in the gPROMS environment. The language supports modeling and simulation of mixed systems of integral, partial and ordinary differential, and algebraic equations (IPDAEs) over rectangular domains. In gPROMS, a domain can be declared as:

```
MODEL TubularReactor

 PARAMETER
   NbrComp            AS INTEGER
   ...
   ReactorLen      AS REAL
   ReactorRad      AS REAL

 DISTRIBUTION_DOMAIN
   Axial         AS ( 0 : ReactorLen )
   Radial        AS ( 0 : ReactorRad )
```

where *Axial* and *Radial* are the independent variables. A dependent variable that is to be solved over this domain is declared as

```
VARIABLE
  Temp  AS DISTRIBUTION (Axial, Radial)
                          OF Temperature
```

Here, *Temp* is the dependent variable of type *Temperature* (defined elsewhere), and it's domain is the three-dimensional area defined by the independent variables *Axial* and *Radial.* This declaration is similar to declaration of arrays in the gPROMS language, which is done as:

```
F   AS ARRAY ( NbrComp )  OF Flowrate
```

For-loops are used for specification of a domain for a specific expression or equation:

```
FOR z:= 0 TO ReactorLen DO
 -Kr*PARTIAL(Temp(z,ReactorRad),Radial)=
  Uh*(Temp(z,ReactorRad)-TWall(z));
 END
```

Here, a boundary condition is defined for the wall of the tubular reactor, which has an axial distribution but not radial. The partial differentiation can also be seen here, which is done with the operator `PARTIAL(expr, var)`, that represents the partial derivative of the expression *expr* with respect to the variable *var* (i.e. $\partial_r Temp(z,R)$). The `PARTIAL` operator supports differentiation of entire expressions, not only single variables.

1.4 MathModelica

MathModelica [9, 8] is a Mathematica-based tool for Modelica-based modeling and simulation. The models are defined interactively in *Mathematica notebooks*, using the Modelica language, either using standard Modelica syntax or using an expression-based syntax similar to Mathematica. Mathematical symbols and expressions can be used in these models, making models easier to read. Figure 1 shows the MathModelica input for a pendulum model. Graphical design of Modelica models based on Modelica library components is also possible in MathModelica.

$$
\begin{array}{l}
Model\ [\ Pendulum, \\
\quad Parameter\ Real\ m == 1, g == 9.82, r == \frac{1}{2}; \\
\quad Real\ \phi\ [\{Start == 0\}], v_t[\{Start == 0\}], a_t, F_t; \\
\quad Equation\ [\\
\qquad F_t == -mgSin[\phi]; \\
\qquad F_t == ma_t; \\
\qquad v_t == r\phi'; \\
\qquad a_t == v_t'; \\
\quad]; \\
]
\end{array}
$$

Figure 1: MathModelica input for a pendulum model. == is the equality operator and ′ is the operator for first derivative of a function of one variable in Mathematica.

The main advantage of using Mathematica as the implementation platform together with MathModelica is that it is very easy to add extensions to the MathModelica language (i.e. Modelica with MathModelica syntax) without modifying the parser which builds the internal syntax tree. The extensions are preserved during this translation, and additional processing can be made after the MathModelica parser, in order to handle the extensions appropriately. This is convenient when experimenting with new language features like the PDE extensions. The transformational programming language that is available in Mathematica is also convenient for handling translation of data structures between different formats.

2 Modelica Extensions

In order to introduce PDE support in Modelica, several issues need to be solved, such as handling spatial variables, domains, and initial and boundary conditions. This chapter presents some ideas regarding these issues.

2.1 Spatial Variables

Variables in Modelica are functions of time, if they are not declared as constants or parameters. In order to introduce spatial variables without changing the language too much, we can introduce a type modifier `space` to be used in declarations of spatial variables. An example can look like this:

```
space Real x, y, z;
```

2.2 Domain Classes

Domain classes are needed to specify the domain where a function is defined or where an equation holds. A variable can be assigned a domain when it is declared, which will then also have its spatial dependencies defined. Several approaches can be taken to design domain classes.

2.2.1 Single Domain Classes

Each domain can be defined using a single domain class. Two new keywords can be introduced, `domain` and `subdomain`, which are language extensions, designating a new kind of restricted class and a new kind of section within such classes. For example, a rectangular domain class ($x \in [0,2], y \in [0,4]$) can be defined as:

```
domain Rectangular
 space Real x(min=0, max=2);
 space Real y(min=0, max=4);
end Rectangular;
```

The attributes *min* and *max* belong to the built-in type `Real`. Lower dimensional sub-domains can be defined in such a domain object by adding a section `subdomain` with an equation or a boolean expression that restricts the domain. For example:

```
domain Rectangular
 space Real x(min=0, max=2);
 space Real y(min=0, max=4);
 subdomain leftright
   x=x.min or x=x.max
 subdomain updown
   y=y.min or y=y.max
end Rectangular;
```

Complex domains can also be defined using expressions as min and max values of the domain variables. For example:

```
domain Circular
 parameter Real r;
 space Real x(min=-r, max=r);
 space Real y(min=-sqrt(r-x^2),
              max=sqrt(r-x^2));
 subdomain edge
   x^2+y^2 == r^2;
end Circular;
```

This approach is used in the prototype described in Section 3.

2.2.2 Composite and Hierarchical Domain Classes

For domains with more complex shapes like circle or polygon, the shape can be defined separately and specified as a boundary for the domain. An example of such a shape is:

```
domain Circle
 parameter Real radius;
 outer space Real x, y;
constraint
 x^2 + y^2 == radius^2;
end Circle;
```

which declares a circle with the center $(0,0)$ and radius 1. The keyword `constraint` is a language extension and specifies the expressions that must be true for the spatial variables to be inside this domain. The reason the existing `equation` keyword is not used is that it should be possible to give boolean expressions as constraints.

Using the shape *Circle* above, a circular domain with a hole can be defined as:

```
domain Ring
inner space Real x, y;
Circle edge1(radius=1);
Circle edge2(radius=3);
constraint
 x^2 + y^2 <= edge2.radius^2;
 x^2 + y^2 >= edge1.radius^2;
end Ring;
```

Several constraints separated by ';' are equivalent to the same constraints combined by the `and` operator. In the future, built-in operators `inside()` and `outside()` might be introduced, in order to specify a relationship between the spatial variables and other domains. In this example, the constraints could be specified using these operators together with the boolean operator `and`:

```
inside(edge2) and outside(edge1);
```

In that case, some restrictions on the referred domains might be required, i.e. that a domain must be a closed curve or a closed polygon in order to be used with these operators.

The domains can also be built using inheritance and composite classes. A complete example might appear as follows:

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

record Point2D
 Real x,y;
end Point2D;

domain Circle2D
 extends Cartesian2D;
 parameter Point2D center(x=0, y=0);
 parameter Real radius=1;
constraint
 (x-center.x)^2 +
 (y-center.y)^2 == radius^2;
end Circle2D;

domain Ring;
 inner space Real x, y;
 Circle2D c1(center(x=0,y=0),
             radius=1);
 Circle2D c2(center(x=0,y=0),
             radius=2);
constraint
 (x-c2.center.x)^2 +
 (y-c2.center.y)^2 <= c2.radius^2;
 (x-c1.center.x)^2 +
 (y-c1.center.y)^2 >= c1.radius^2;
end Ring;
```

With the `inside()` and `outside()` operators, the constraints for the *Ring* domain would be

```
constraint
 inside(c2) and outside(c1);
```

2.3 Component Declarations

Once domains are defined, variables that depend on spatial variables, called distributed variables, can be declared and assigned a domain object. A model with a variable *U* defined over the ring-shaped domain (see Section 2.2) can be specified as:

```
model PDETest
  Ring ringdom;
  Real U(domain=ringdom);
  ...
end Test;
```

The attribute `domain` is an addition to the existing type `Real` in Modelica[1], not to be confused with the previously introduced use of the keyword `domain` as a restricted class. Modification of the `domain` attribute of the type `Real` declares the definition domain of *U*. This modification is also an implicit way of declaring *U* to be a spatially dependent variable, besides being time-dependent. Alternatively, declaring a variable to be spatially dependent can be done explicitly using a new keyword, for example the keyword `field`, or the keyword `dependent`:

```
depdendent Real U(domain=ringdom);
```

2.4 Domain Specification for Equations

When specifying an equation, i.e. the PDE or one of the boundary conditions, the domain over which the equation is valid needs to be specified. One way to do this is to use for-loops. With a syntax similar to Modelica, this can be done as:

```
equation
  for (x,y) in ringdom loop
    pder(U,x,2) = sin(x)+cos(y);
  end for;
```

Here, a new built-in operator `pder(U,x,i)` is also introduced, which represents the i:th derivative of *U* with respect to *x*. Instead of using the component *ringdom* directly, one could refer to the domain of *U* using dot notation, e.g. *U.domain*.

The difference between for-loops in Modelica and the for-loop in the example above is the usage of multiple variables as loop indexes and a domain object as a range[2]. The scope of the variables *x* and *y* is local for the loop, and they are aliases for the spatial variables in the domain, with *x* corresponding to the first declared space variable in the domain and *y* corresponding to the second.

An alternative syntax for domain specification is

```
pder(U,x,2) =
   sin(x) + cos(y), (x,y) in dom;
```

which is closer to mathematical notation. Both alternatives can be supported as well, with one being syntactic sugar for the other.

Another, somewhat less convenient alternative is to use the spatial variables in a domain object directly, using the member notation:

```
pder(U,dom.x,2) =
  sin(dom.x)+cos(dom.y);
```

This could also co-exist with the other alternatives.

2.5 Initial and Boundary Conditions

Initial values for variables in Modelica are given using the `start` attribute of the built-in types. Similarly, initial conditions for a PDE problem can be given as modification of the `start` attribute of distributed variables. The example from Section 2.3 with initial conditions added looks like:

```
parameter Real f(domain=ringdom);
 Real U(domain=ringdom, start=f);
```

Here, a parameter *f* is declared, which can be given as argument to the simulator, or defined in the model as:

```
parameter Real f(domain=ringdom) =
  sin(x) + cos(y), (x,y) in ringdom;
```

The expression in the right-hand side can be given as a value to the *start* attribute of *U* directly, as well.

Boundary conditions can be specified using the specific sub-domains of a domain in the equations defining the boundary conditions. A new keyword `boundary` is used to separate model equations from boundary conditions, e.g.:

```
boundary
  for (x,y) in ringdom.c1 loop
    U(x,y) = -5;
  end for;
```

This defines the value of *U* on the inner circle of the ring domain, declared as *c1* in *Ring*.

3 Current Work

A prototype translator has been implemented in Mathematica, using the MathModelica environment (see Section 1.4). This translator supports the kind of domain classes described in Section 2.2.1, except that only single equations are recognized in the subdomains and not boolean expressions. An example defining a circular domain can be seen in Figure 2.

Domain references in equations are done by directly putting the domain object as an argument to the dependent variable. Thus, all dependent variables have two arguments, the domain object which implicitly defines all spatial arguments, and the time variable.

[1] The regular type Real can have a built-in one-dimensional domain $[min, max]$ as default, to be consistent with current Modelica.

[2] Ranges in Modelica are written as, for example, 1:10 or 1:2:10 which mean 1,2,3,... or 1,3,5,... , respectively.

```
Domain [ Circular,
  Parameter Real r == 1;
  Space Real x[{min == -sqrt(r^2 - y^2),
                max == sqrt(r^2 - y^2)}];
  Space Real y[{min == -r, max == r}];
  Subdomain [ leftborder,
    Equation [ x == x.min ];
  ];
  Subdomain [ rightborder,
    Equation [ x == x.max ];
  ];
]
```

Figure 2: A domain declaration in the prototype translator, defining a circular domain

For example:

```
Equation [
  ∂t U[circdom, time] - ∂xx U[circdom, time] == 0;
];
```

Boundary and initial conditions are put into a section called `Boundary[]` as normal equations. In boundary condition equations, a sub-domain of a domain is referred with dot notation and used as the first argument. In initial conditions, the time argument is set to a constant. For example:

```
Boundary [
  ∂t U[circdom.leftborder, time] == 0;
  U[circdom.leftborder, time] == 2;
  U[circdom, 0] == Sin[circdom.x] + Cos[circdom.y];
];
```

The translator uses MathModelica for parsing the input, and modifies the result for the additions that are done to the syntax for PDE support. Then, the PDE, boundary and initial conditions, and the geometry of the model are extracted and converted to an input format that is used by the solver, a PDE solver [13] implemented in Mathematica. This solver uses the finite difference method to generate special C++ code for the given problem and the selected differentiation methods.

3.1 Example

As an example, we can define an initial value problem with a two-dimensional wave equation using a circular domain. The equation looks like

$$\partial_{tt}u = \partial_{xx}u + \partial_{yy}u, \qquad x^2 + y^2 <= 1$$

with the initial and boundary conditions

$$\begin{array}{ll} u = 9cone(x,y,0.4)sin(\pi x)cos(\pi x), & t = 0 \\ \partial_t u = 0, & t = 0 \\ u = \delta, & x^2 + y^2 = 1 \\ \partial_x u = 0, & x^2 + y^2 = 1 \\ \partial_y u = 0, & x^2 + y^2 = 1 \end{array}$$

where $cone(x,y,r)$ is defined as

$$cone(x,y,r) = \begin{cases} 1 - \frac{\sqrt{x^2+y^2}}{r^2} & \sqrt{x^2+y^2} <= r \\ 0 & otherwise \end{cases}$$

First, we define a domain object representing a general circular domain class:

```
Domain [ Circle
  Parameter Real xc == 0;
  Parameter Real yc == 0;
  Parameter Real r == 1;
  Space Real x[{min == xc - sqrt(r^2 - (y - yc)^2),
                max == xc + sqrt(r^2 - (y - yc)^2)}];
  Space Real y[{min == yc - r, max == yc + r}];
  Subdomain [ leftborder,
    Equation [ x == x.min ];
  ];
  Subdomain [ rightborder,
    Equation [ x == x.max ];
  ];
];
```

The default values specify a circular domain with the center $(0,0)$ and the radius 1 when the domain is instantiated. Then we can specify the main model:

```
PDEModel [ TestModel
  Parameter Real xc == 0;
  Parameter Real yc == 0;
  Parameter Real r == 1;
  Parameter Real delta == 0;
  Circle dom[{xc == xc, yc == yc, r == r}];
  Dependent Real u;
```

```
Boundary [
  u[dom,0]== uinit[dom.x,dom.y];
  ∂t u[dom,0]== 0;
  u[dom.leftborder,time]== delta;
  ∂x u[dom.leftborder,time]== 0;
  ∂y u[dom.leftborder,time]== 0;
  u[dom.rightborder,time]== delta;
  ∂x u[dom.rightborder,time]== 0;
  ∂y u[dom.rightborder,time]== 0;
];

Equation [
  ∂tt u[dom,time]==
    ∂xx u[dom,time]+∂yy u[dom,time];
];
];
```

The keyword `PDEModel` is used by the translator to recognize PDE models. The start values of the dependent variable *u* are given as a function *uinit[x,y]* defined in the Mathematica environment as:

$$uinit[x\_,y\_] = 9MyCone[x,y,0.4]Sin[\pi x]Cos[\pi y];$$

MyCone generates a cone with the center $(0,0)$ and a given radius, and is used to limit the *Sin* and *Cos* functions to a small area of the domain. A plot of the initial values with radius set to 0.4 can be seen in Figure 3).

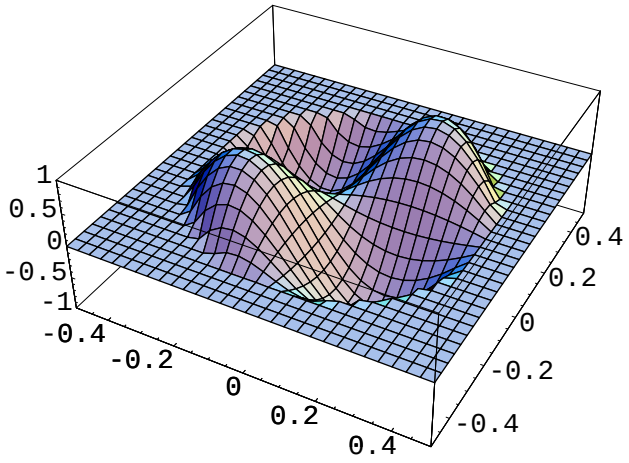


Figure 3: Initial value function *uinit* with *radius* = 0.4 for the PDE used in TestModel.

Then, the input to the PDE solver mentioned in Section 3 can be generated using the function *GenerateSolvePDEInput[],* which takes as argument the name of the model and some parameters needed by the solver, such as the differentiation methods, start and end values for the time variable and the spatial variables, and size of the result array. In this example, we use the intervals $x \in (-1,1)$, $y \in (-1,1)$, $t \in (0,0.25)$, a 100x100 grid and 200 time steps. Figure 4 shows the result at $t = 0.0625$, 0.125 and 0.1875.

4 Conclusion and Future Work

The purpose of this paper is to consider some initial extensions to the Modelica language that are needed in order to allow modeling with partial differential equations. The focus is the definition of the domain objects and how they are used together with equations and initial and boundary conditions.

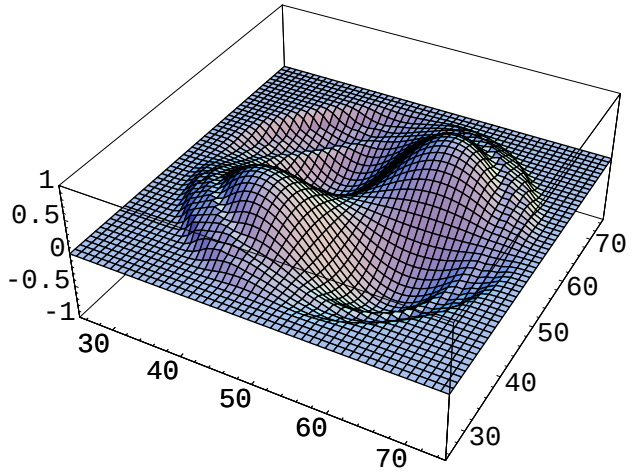
Future work involves further implementation of the extensions discussed in Section 2. Also, an extended syntax for domains will be designed, to allow parametric specification of boundaries. Another extension that can be needed is modification of the `function` type in Modelica for spatially distributed variables.

A powerful feature of Modelica is the `connect()` statement, that allows building models using smaller submodels. Extension of this feature for PDE models needs to be considered, as well. Possible approaches are using solvers that support domain-decomposition techniques, or if different PDE models need to be connected, techniques like the *interface relaxation method* [11].

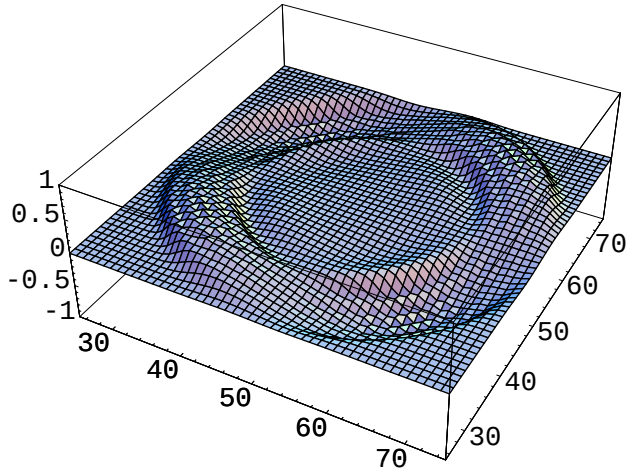
For solving the PDE problems, specification of grids, triangulations or triangulation algorithms to be used with finite element methods and solution method selection (e.g. selection of finite difference methods for the solver mentioned in Section 3) needs to be included in Modelica, or defined in a separate support language.

Furthermore, automatic solver selection can be considered, for example by using pattern matching on equations and selecting a solver or method from a database. An architecture with several solvers and the possibility to add new solvers would make a useful environment.

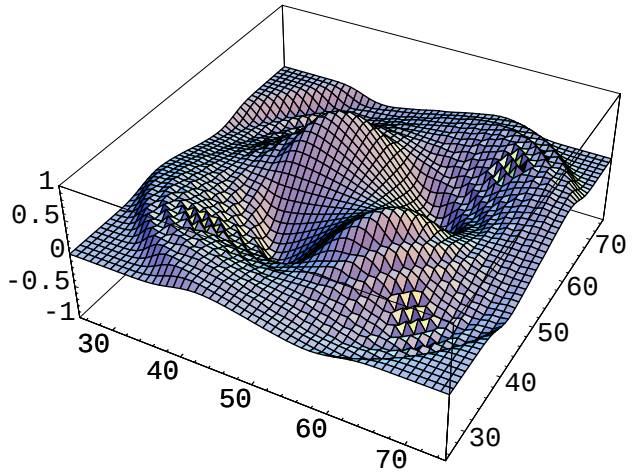
Another problem is adapting the PDE solvers and existing DAE solvers that are used for the existing ODE-based models, or finding other ways to support solution of existing models together with PDE models.



$t = 0.0625$



$t = 0.125$



$t = 0.1875$

Figure 4: Plot of the solution at $t = 0.0625$, $t = 0.125$ and $t = 0.1875$.

5 Acknowledgments

This work has been supported by the ECSEL graduate school at Linköping University, supported by the Swedish Strategic Research Foundation.

References

[1] Modelica Association 1999. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 1.3*, Dec 1999.

[2] A. F. Cardenas and W. J. Karplus. PDEL — a language for partial differential equations. *Communications of the ACM*, 13(3):184–191, March 1970.

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

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

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

[6] Hitachi, Ltd., Computer Division. *PDEQSOL User's Manual*, 1990.

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

[8] M. Jirstrand. MathModelica, a full sytem simulation tool. In *Proc. of Modelica Workshop 2000*, 2000.

[9] M. Jirstrand, J. Gunnarsson, and P. Fritzson. MathModelica – a new modeling and simulation environment for Mathematica. IMS '99—Third International Mathematica Symposium, 1999.

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

[11] J. R. Rice. An agent-based architecture for solving partial differential equations. *SIAM News*, 31(6), 1998.

[12] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems using ELLPACK*. Springer-Verlag, 1985.

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

[14] S. Weerawarana. *Problem solving environments for partial differential equation based applications.* PhD thesis, Department of Computer Sciences, Purdue University, August 1994.