PDE extension
Changes over Levon's extension

Jan Šilar
jan.silar@lf1.cuni.cz

October 16, 2014

New extension is compared to Levon's work ([2]), mostly chapter 4

Domains Geometry Definition

Originally
see [2] -- 4.3.1.1 and 4.3.1.2

Saldamli defines domain shape by listing its boundaries. Individual boundaries (points in 1D, curves in 2D resp. surfaces in 3D) are describes by shape-functions. Shape-function maps intervals ([0,1] for curves, [0,1]×[0,1] for surfaces) onto the boundary.

Example half-circular domain according to [2]:

```
class Arc
  extends Boundary(ndims=2);
  parameter Point c = {0,0};
  parameter Real r = 1;
  parameter Real a_start = 0;
  parameter Real a_end = 2*Pi;
  redeclare function shape
    input Real tau;          //tau in [0,1]
    output Real coord[2];
  algorithm
    coord := c + r * { cos(a_start + (a_end - a_start) * tau),
                       sin(a_start + (a_end - a_start) * tau) };
  end shape;
end Arc;
```

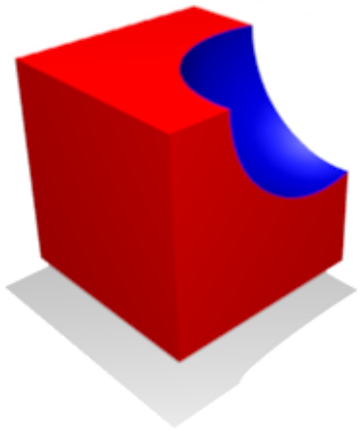

Figure 0.1: Boundary in 3D

```
class Line
  extends Boundary(ndims=2);
  parameter Point p1 = {0,0};
  parameter Point p2 = {1,0};
  redeclare function shape
    input Real h;          //h in [0,1]
    output Real coord[2];
  algorithm
    coord := p1 + (p2 - p1) * h;
  end shape;
end Line;

type Half-circularDomain
  extends Cartesian2D(boundary = {arc, line});
  parameter Arc arc (c = {0,0}, r = 2, a_start = Pi/2, a_end =
Pi*3/2);
  parameter Line line (p1 = {0,-2}, p2 = {0,2});
end Half-circularDomain;
```

Problem

This approach doesn't work well in 3D: if boundaries of the whole domain are composed of several surfaces, parameters (arguments) of shape-functions of these surfaces must be bounded not just in $[0, a] \times [0, b]$ interval but in some more complex set for each boundary surface so that they form a continuous boundary, e.g. see fig. 0.1. And there is no way to write this in Levon's extension.

Even if the syntax allowed this, it would be difficult for the user to determine these sets where parameters are bounded.

There is also no simple way to generate grid points during translation/solution.

Alternative approach We define explicitely both interior and boundaries of the domain (these elements are called *regions* here).

We have new built-in type `Coordinate`:

```
type Coordinate = Real;
```

Another new built-in type is `Domain`. It contains also `type Region` to represent interior and boundaries of the domain. `Region` is nested in `Domain` to prevent instantiating `Region` outside a `Domain`.

```
type Domain
  type Region
    parameter Integer ndim; //dimension of the region
    parameter Real[ndims][2] interval;
  end Region;
  replaceable function shapeFunc
    input Real u[ndim];
    output Real coord[ndim];
  end shapeFunc;
  parameter Integer ndimD; //dimension of the domain
  Coordinate coord[ndimD];
  replaceable Region interior; //main region of the domain
end Domain;
```

`type Domain` is extended by domain types for particular dimensions, e.g. in 2D:

```
type Domain2D
  extends Domain(ndimD = 2);
  type Region0D = Region(ndim = 0); //for points
  type Region1D = Region(ndim = 1); //for boundaries
  type Region2D = Region(ndim = 2); //for interior
end Domain2D;
```

These types are extended by particular domains, e.g.:

```
type DomainRectangle2D
  extends Domain2D;
  parameter Real lx = 1; //length in x dir.
  parameter Real ly = 1; //lenght in y dir.
  Coordinate x(name = "cartesian") = coord[1]; //alias for the first coordinate
  Coordinate y(name = "cartesian") = coord[2]; //alias for the second coordinate
  Region2D interior(x in {0,lx}, y in {0,ly});
  Region1D top(x in {0,ly}, y = ly); //boundaries
  Region1D right(x = lx, y in {0,ly});
  Region1D bottom(x in {0,ly}, y = 0);
  Region1D left(x = 0, y in {0,ly});
end DomainRectangle2D;
```

New syntax (`x in {0,lx}`) is used here to specify the interval for the coordinate x within the region.

If the domain geometry is more complex than just cartesian product of intervals we define all regions of the domain using one common *shape-function* (or assignment or equation) and for each region we

specify intervals for the shape-function arguments (idea from Peters Book [1]). This approach isn't more general (actualy less), but is consistent in 1, 2 and 3D and (to me) si more natural. Example half-circular domain:

```
type DomainHalf-circle         //1.  variant
  extends Domain2D;
  Coordinate x(name = "cartesian") = coord[1];
  Coordinate y(name = "cartesian") = coord[2];
  parameter Real radius = 2;
  parameter Real[2] c = {0,0};
  Region2D interior(interval = {{0,1},{0,1}});
  Region1D arc(interval = {1,{0,1}});
  Region1D line(interval = {{-1,1},0});
  redaclare function shapeFunc
    input Real r,v;
    output Real coordinate[2];
  algorithm
    coordinate := c + radius * r * { cos(Pi*(1/2 + v),
                                     sin(Pi*(1/2 + v) };
  end shapeFunc;
end DomainHalf-circle;
```

Shape-function is not a pure function (here parameters `c` and `radius` are defined outside the function body) and thus it is not supported in current Modelica. The shape-function may be replaced by just algorithm inserted directly into the domain class to avoid this problem. Parameters of shape-function are replaced by new general auxiliary coordinate system (`r,v` here) (that may not have a good physical meaning in some cases):

```
type DomainHalf-circle        //2.  variant
  extends Domain2D;
  Coordinate x(name = "cartesian") = coord[1];
  Coordinate y(name = "cartesian") = coord[2];
  parameter Real radius = 2;
  parameter Real[2] c = {0,0};
  Coordinate r,v;
  Region2D interior(r in {0,1}, v in {0,1});
  Region1D arc(r = 1, v in {0,1});
  Region1D line(r in {-1,1}, v = 0);
algorithm
  coord := c + radius * r * { cos(Pi*(1/2 + v), sin(Pi*(1/2 + v) };
end DomainHalf-circle;
```

Probably the algorithm is not needed and may be replaced by equations:

```
type DomainHalf-circle        //3.  variant
  extends Domain2D;
  Coordinate x(name = "cartesian") = coord[1];
  Coordinate y(name = "cartesian") = coord[2];
  parameter Real radius = 2;
  parameter Real[2] c = {0,0};
  Coordinate r,v;
  Region2D interior(r in {0,1}, v in {0,1});
  Region1D arc(r = 1, v in {0,1});
  Region1D line(r in {-1,1}, v = 0);
equation
  coord = c + radius * r * { cos(Pi*(1/2 + v), sin(Pi*(1/2 + v) };
end DomainHalf-circle;
```

One of these approches should be chosen. I prefare the third one.

More complex geometries

May be more complex geometries could be defined using *Constructive Solid Geometry* -- it is applying union, intersection and difference on previously defined shapes. The syntax is not designed already. It should be also possible to define domain in external file from some CAD app.

Differential operators

4.3.2

Partial derivatives

Originally
see 4.3.2.1
e.g. $\frac{\partial u}{\partial t}$ .. `der(u)`, $\frac{\partial^2 u}{\partial x \partial y}$ .. `der(u,x,y)`

Further specification
for higher order time derivative and mixed time and space derivative, we write `time` explicitly, e.g.
`der(u,time,time)` for $\frac{\partial^2 u}{\partial t^2}$
and
`der(u,x,time)` for $\frac{\partial^2 u}{\partial x \partial t}$.

Normal derivative and normal vector

Originally
see 4.3.2.2

normal vector is implicit member of domain

Problem
Normal vector makes sense only in regions of dimension *n-1* in *n*-dimensional domain (i.e. surface in 3D, curve in 2D and point in 1D). There is no normal vector in *n* dimensional region and infinitely many in less than *n-1* dimensional regions.

Alternative approach
normal vector `n` is implicit member of all *n-1* dimensional regions in *n*-dimensional domain. So we write

```
pder(u,omega.boundary.n) = 0 in omega.boundary;
```

A shorten notation is suggested in next section.
Using normal vector outside differential operators should be also possible e.g.:

```
field Real[3] flux;
flux*omega.boundary.n = 0 in omega.boundary;
```

Accessing coordinates and normal vector in der() operator

Originally
not discussed

Problem
Coordinates and normal vector are defined within the domain class, but they are used in equations that are written outside domains. Thus they should be accessed using `domainName.` prefix (e.g. `omega.x`), which is tedious.

In the example in 4.3.2.2 in [2] the normal vector `n` is reached outside the domain class without `domainName.` prefix even thou it is defined in the domain. It is not explained how this is enabled.

Solution
Fields are differentiated with respect to coordinates or normal vector only (or may be also some other vector for directional derivative??). Thus in place of second and following operands of `der()` operator may be given only coordinates or normal vector. So variables in this positions may be treated specially and coordinates and normal vector of the domain of the field being differentiated may be accessed without the `domainName.` prefix here.

If coordinates or normal vector is used in different context (not in place of second and following operands of `der()`), an alias for it may be defined in the model, e.g.

```
Coordinate x = omega.x;
```

Perhaps usage of this shortened notation was intended even in the original extension but was not mentioned.

Start values of derivatives

Originally
not discussed

problem
Higher derivatives are allowed for fields thus we need to assign initial values to its derivatives sometimes.

solution
New attributes `startPrime` and `startSecond` (May be `startSecond` is not needed??) for field variables are introduced. Usage e.g.:

```
field Real u(start = 0, startPrime = sin(omega.x*omega.y));
```

Initial values for higher derivatives must be assigned in `initial equation` section.

`In` operator

Just a remark: All equations containing a field variable (defined on a domain) hold on particular region of the domain. If the region is not specified (using `"in domain.region"`) region `interior` is assumed implicitly.

Accessing field values

Originally
see 4.2.4, in function-like style

problem
It is not consistent with curent Modelica -- to access values of regular variables in particular time in this function-like style is also not allowed.

If more then one coordinate system are defined in a domain (discussed later), it is not clear which coordinates are used in the function-like expression.

solution
Regions consisting of one point and the `in` operator will be used instead to represent the particular point. E.g.

```
model heatPID
  record Room extends DomainBlock3D;
    Region0D sensorPosition(shape = shapeFunc, range = {{1, 1}, {0.5, 0.5}, {0.5,
0.5}});
  end Room
  Room room(...)
  field Real T(domain = room);
  Real Ts;
  ...
equation
  Ts = T in room.sensorPosition;
  ...
end heatPID;
```

`in` operator will be probably used also to match regions from different domains and to write equations (boundary conditions) relating fields from different domains. The syntax is not developed yet.

Modifications presented below are not so important and are questionable.

Coordinates

Originally
see 4.3.1.1 and 4.3.1.3

There are two arrays for coordinates predefined in the built-in `Domain` type. `cartesian` for cartesian coordinates and `coord` for arbitrary coordinates specified by the user. No other coordinates may be defined except aliases to elements of these predefined arrays.

Problem
May be this is not flexible enough. User may need more different coordinate systems.

Solution
new type `Coordinate` to define coordinates. Usage e.g.

```
Coordinate coordName;
```

The array `coord` in the built-in `Domain` type may be left out then.

Field literal constructor

originally
see 4.2.2, e.g.:

```
u = field(2*a+b for (a,b) in omega)
```

where iterator variables `(a,b)` exist only in constructor expression and represent coordinates in omega (probably `coord`, but may be `cartesian`, it is not clear from the document.)

problem
This syntactic feature is redundant (explained in next paragraph). Another problem is that it enables to define the field values in terms of only one coordinate system. There may be two (or more) coordinate systems defined and it may be useful to be able to define fields using any of them.

solution Coordinate variables are special kind of fields. (Coordinates vary their value over space as other fields.) Thus any expression depending on coordinates should be evaluated to another field. So we can write just

```
u = 2*omega.x+omega.y;
```

According to the section "In operator" this is equivalent to `u = 2*omega.x+omega.y in omega.interior;`

If coordinates are used often (not only in `pder()`), an alias for it may be defined to avoid `"omega."` prefix, e.g.

```
Coordinate x = omega.x;
```

Further problems

How to write equations (mainly BC) connecting fields defined in different domains (some kind of distributed connectors)?

References

[1] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, 2004.

[2] Levon Saldamli. *A High-Level Language for Modeling with Partial Differential Equations.* PhD thesis, Department of Computer and Information Science, Linköping University, 2006.