Curvilinear Coordinates in a PDE Solver Framework; Analysis

Malin Ljungberg

August 20, 2004

Abstract

In the present work we demonstrate how the use of an object oriented analysis and design process, augmented by variability modeling and user centered design, results in a flexible software model for handling curvilinear coordinates in a finite difference PDE solver framework. It is shown that feature and variability modeling are appropriate tools for identifying and achieving the appropriate flexibility in this model.

The resulting software model is validated through an comparative analysis involving a range of other similar models.

1 Introduction

As scientific codes become increasingly complex there is a growing recognition of the impossibility of incorporating all the necessary functionality in one single package. There are several efforts aimed at the creation of frameworks for the simulation of physical phenomena. In this context the term "framework" can have a range of interpretations. It can mean anything from a software library to a set of interoperable stand alone programs. Here we use the term *framework* in a wide sense, meaning a set of interoperable software modules which can be put together in different ways in order to perform different tasks.

It has been recognized that in the construction of numerical solvers for Partial Differential Equations (PDE), increased flexibility can be achieved by using such a modular design. By implementing a well designed library framework rather than just writing each specific PDE solver separately a reduction can be obtained of the total time and effort involved in developing as well as maintaining and updating the PDE solvers thus created.

A prerequisite for such a framework is that it contains the parts necessary in order to build a wide range of PDE solvers, and that it is constructed in such a way that it is easy to add functionality that is not present from the start. Another prerequisite is that the efficiency of PDE solvers built from the framework is on a par with that of PDE solvers built from scratch.

Finding a suitable architecture for a framework for numerical solution of PDE using finite difference methods on structured grids, has been an active line of research for our group since the early 1990's, (see [HT92] [ORT96] [MOT97] [Åhl99a] [TÅL+01]). The idea is that a framework should consist of an appropriate set of reusable software components. The construction of a particular PDE solver should be achieved by selecting and combining suitable components from the framework, and possibly adding functionality in the form of a new component to be incorporated in the framework. Earlier work in our group has resulted in two separate prototype frameworks. Compose, [Åhl99b], and Cogito [Tea97]. More recently the TENGO (TEnsor Notation for Grid Operators) framework [ÅO04], has been developed. This is a set of Fortran 90 modules representing discretized tensor fields and operations on these.

The general issue of a "PDE Solver Framework" is very broad. In order to identify appropriate and manageable research issues it is necessary to focus in on a more narrow setting. We have chosen the subset of finite difference PDE solvers, that use structured grids placed along the coordinate lines of curvilinear coordinate systems. This type of solver allows for efficient numerical solution algorithms, but requires relatively sophisticated software for the numerical approximation of the PDEs, due to the use of curvilinear coordinates. This expected complexity motivates careful investigation of possible software architectures for such PDE solver frameworks.

With TENGO we have an *infrastructure* for the kind of PDE solver framework we are studying. Also, we consider the Compose framework to be an appropriate *superstructure* for such frameworks. Consequently, the focus of our present investigations is on the new framework components that are needed *in between* those two framework layers.

In the following, an important objective is to carry out an *analysis* of how to structure the parts of a PDE solver framework that are related to the handling of curvilinear coordinates. The analysis serves to identify exactly *what* those parts are, and what kind of variation this framework subset should be able to support within those parts. Moreover, comparisons will be made between different ways of structuring the relevant parts of the framework, in order to achieve the required variation. This analysis, together with the resulting model, comprise the main results of the present paper.

In an accompanying article [LO04] we investigate how flexibility requirements can be combined with other requirements, particularly with respect to performance. Often a design change which results in an increase of efficiency will result in a decrease of flexibility, and vice versa. In [LO04] we validate that both these goals are met by the framework subset proposed here.

Unlike many of the existing PDE solver frameworks, such as for example [PS02] [Gra02] [KeL01] [GRA] [hyp01] [PET02] [Hor00] [POO99] [Ove99] [HFJ99] [Tea97], the present work demonstrates the process by which the suggested software model is achieved. Projects which have also demonstrated the process by which the software model was developed are [BL97] [HW99] [LS99] [Åhl99b] [DBQ99] [CZ01] [HK02] [Hav00]. All of these use an object oriented analysis and design process, and the last two use design patterns and algebraic modeling respectively, as well. In the present work we demonstrate how the use of such an object oriented analysis and design process, augmented by variability modeling and user centered design, results in a flexible software model for handling curvilinear coordinates in a PDE solver framework.

2 Curvilinear structured grids

The concepts involved when using curvilinear structured grids in a finite difference PDE solver are discussed here as a basis for the feature and variability modeling presented in Section 3.

The idea of having a mesh or a grid mapped onto some physical region is fundamental to the types of PDE solvers we are looking at. A structured grid can be constructed by laying the grid points out evenly along the coordinate lines of some coordinate system in the physical region. If this coordinate system is Cartesian we get an equidistant rectangular grid. If we want a rectangular grid which is not equidistant we need to allow for scaling of the coordinate axes, which is equivalent to laying the grid points out unevenly along the coordinate lines. This is an approach commonly used when modeling boundary layer phenomena, for example.

If we put the grid on some coordinate system other than Cartesian we get what is called a curvilinear structured grid.

It should be noted that one strong argument for using a curvilinear structured grid is that it may allow for a very convenient treatment of the boundaries, provided that the curvilinear coordinate system can be picked to agree with the geometry of the modeled region. For example, if modeling the flow through a cylindrical pipe using a cylindrical coordinate system, one may use the fact that the wall of the pipe is a surface of constant radius, r. Below we will give a short summary of concepts used to describe curvilinear structured grids, closely following [KS94].

If we look at a curvilinear coordinate system without a grid laid upon it, we can start out by considering the coordinates. Any point in physical space can be described by a vector pointing to it. This constitutes a vector field, which we will refer to as $\boldsymbol{x}$. The components of this vector field can be found by projecting the vector onto Cartesian basis vectors. If we use a Cartesian coordinate system the components will simply be x_i, but if we instead describe the vector using curvilinear coordinates, ξ_i, then the components will be expressions in those curvilinear coordinates. As an example we have that the vector components of a point described in cylindrical coordinates are $(r\sin(\theta), r\cos(\theta))$. Another way of viewing the relationship between the curvilinear and the Cartesian coordinates is to see the vector field as the result of a mapping from a flat coordinate space, the *logical space*, onto a vector field, the *physical space*. We call this the coordinate mapping. Figure 1 illustrates the flat (r, θ) space together with the space onto which it is mapped.

The coordinates ξ_i describe positions in a coordinate space, that has a different curvature than the true physical space. This means that a small movement along a straight line in physical space, will generally correspond to a movement along a curve in logical space, and vice versa. The curves in physical space which correspond to motion along one of the coordinates while keeping the others fixed are called coordinate lines. If the coordinate lines intersect at right angles we call the coordinates orthogonal.

It should also be noted that the physical space may have a higher dimensionality than the logical space. A piece of a spherical surface can be described by two angular coordinates, even though points on it are described by three Cartesian coordinates in physical space.

When expressing differential operators in curvilinear coordinates the fact that the logical space is curved relative to the physical space comes into play. The metric tensor, $\mathcal{G}$, the components of which can be found as $\mathcal{G}_{i,j} =$

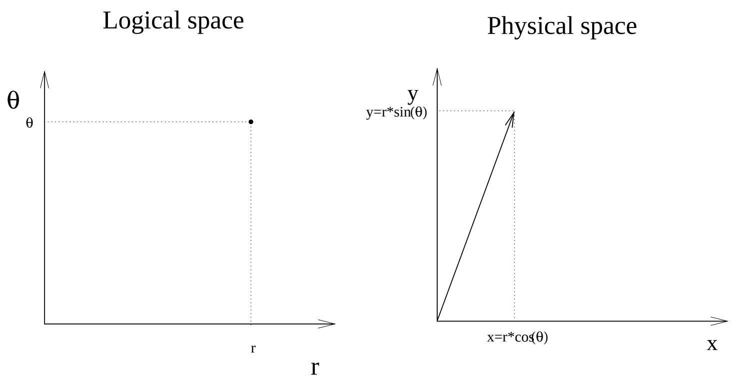

Figure 1: Example of curvilinear coordinates.

$\dfrac{\partial \boldsymbol{x}}{\partial \xi^i} \cdot \dfrac{\partial \boldsymbol{x}}{\partial \xi^j}$, describes this curvature. These components will have to be included when writing the PDE on component form, to compensate for the fact that the infinitesimal changes now are taking place along the curved coordinate lines. It should be noted that the metric tensor is diagonal for orthogonal coordinates. For the example of cylindrical coordinates we get that

$$\mathcal{G}_{1,1} = 1$$
$$\mathcal{G}_{2,2} = r^2,$$

where r is the first coordinate and θ the second.

The cylindrical coordinates are orthogonal, so the off-diagonal elements of $\mathcal{G}$ are zero.

If we for example express the gradient of some scalar Φ in cylindrical coordinates we get

$$\nabla\Phi = \frac{\partial \Phi}{\partial r}\hat{\boldsymbol{r}} + \frac{1}{r^2}\frac{\partial \Phi}{\partial \theta}\hat{\boldsymbol{\theta}},$$

where $\hat{\boldsymbol{r}}$ and $\hat{\boldsymbol{\theta}}$ are the coordinate vectors defined as $\frac{\partial \boldsymbol{x}}{\partial r}$, and $\frac{\partial \boldsymbol{x}}{\partial \theta}$ respectively. If we choose to work with normalized coordinate basis vectors, $\hat{\boldsymbol{e}}_{\boldsymbol{i}}$ the expression instead becomes

$$\nabla\Phi = \frac{\partial \Phi}{\partial r}\hat{\boldsymbol{e}}_{\boldsymbol{r}} + \frac{1}{r}\frac{\partial \Phi}{\partial \theta}\hat{\boldsymbol{e}}_{\boldsymbol{\theta}},$$

since one of the factors of r cancels with the magnitude of $\hat{\boldsymbol{\theta}}$.

With this understanding of coordinate systems we can go on to looking at what happens when we put a grid on a curvilinear coordinate system. When we say that the grid is structured we mean that the position of each grid point in physical space is easily deduced from its indices, in a consistent way. In our applications we limit ourselves to the case where a certain value for an index along one of the coordinate directions corresponds to a certain value of

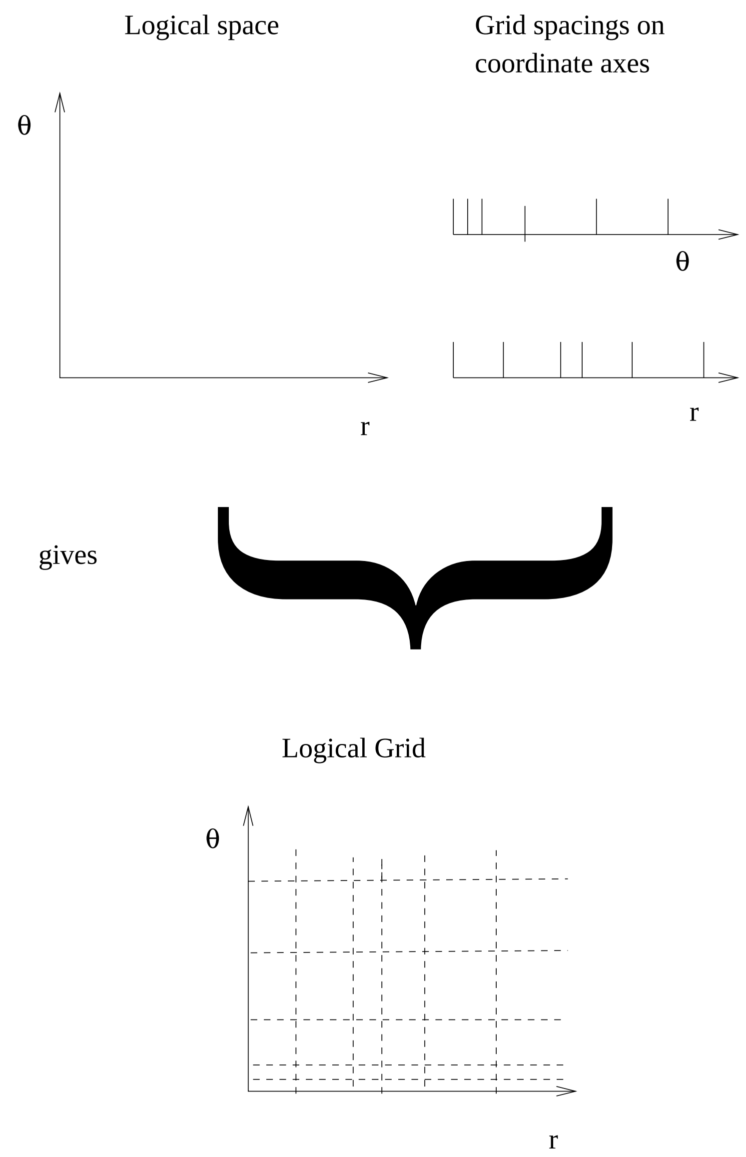

Figure 2: Putting a grid on the logical coordinate space.

the corresponding coordinate, independently of the other coordinates. This represents a rectangular grid in logical space, as is illustrated in Figure 2.

Every point in logical space corresponds to a position in physical space. Thus the entire grid, the *logical grid*, maps onto a corresponding grid in physical space, the *physical grid*, as can be seen in Figure 3. Furthermore, since lines of constant indices for the grid are also lines of constant coordinate values, they will lie on the coordinate lines in physical space. The Cartesian components of any grid point in physical space can be found using the coordinate mapping described above. The components of the metric tensor can also be found, provided that the derivatives $\dfrac{\partial \boldsymbol{x}}{\partial \xi^i}$ are known. Since the solution to the PDE is computed only on the grid, these quantities are needed only at the grid points.

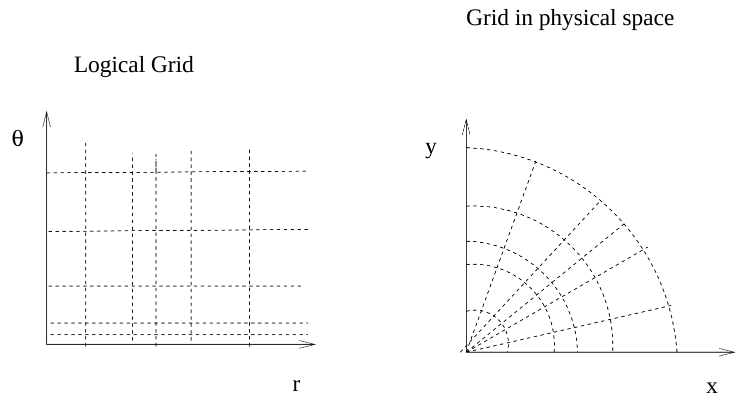

Figure 3: A structured grid based on curvilinear coordinates.

In scientific computing it is common that the structured grids that are used are not easily described analytically. They are generated by so called grid generators, which give the Cartesian components of the coordinates of the grid points, x_i, and sometimes also the derivatives $\dfrac{\partial \boldsymbol{x}}{\partial \xi^i}$ or some derived quantity, such as for example the non-zero components of the metric tensor. These derivatives may have been computed from some analytical expression used by the grid generator, or they may be numerical approximations based on the coordinate values.

It is common practice to let the *unit square* in logical space map onto the particular geometry in physical space that is modeled (cf. [KS94], chapter 5). Sometimes grid points are introduced which lie outside of this region, due to the computational methods used. This corresponds to grid points outside of the unit square in logical space.

Due to symmetries of the problem to be solved it may suffice to lay the grid out along some of the logical coordinates. A problem defined in an axial symmetry would, for example, only need to be gridded along the r and z axes of a cylindrical coordinate system, since the solution to the problem is independent of the angle.

The boundary conditions of the continuous problem will be formulated in terms of spatial operators $\mathcal{P}_i$ applied on one or several of the unknowns of the problem. The discretization of these operators will generally have to be done differently than the discretization of the operators in the inner part of the domain, since points are not available on both sides of the boundary. If $\mathcal{P}_i$, for some i, involves only first derivatives, so called Neumann boundary conditions, it is customary to pick a discretization which has its last point at a distance $h/2$ outside of the boundary, and its second to last point at a distance $h/2$ inside of the boundary. We refer to this as a *boundary centered* discretization. This type of discretization makes it possible to use data at fewer points when approximating the derivative than with a conventional layout of the points, while achieving the same accuracy.

Figure 4: Boundary centered discretization.

If a boundary condition is of Dirichlet or mixed Dirichlet-Neumann type, then it may be advantageous to discretize the corresponding boundary in such a way that the last grid point is located exactly on the boundary. We call this *on-boundary* discretization.

For some PDEs a more accurate solution is achieved by discretizing parts of the equations on grids which are *staggered* with respect to the *base grid* (for a well known example see [Yee66]). For these types of discretizations metric coefficients will be needed also at the grid points of the staggered grids.

In the type of solvers we are looking at the approximation of the PDE is done through discretization on a structured grid. The result of this process will be a matrix equation on the form

$$Bu = y,$$

where u is a vector which represents the unknown quantity on the grid, and B and y come from the discretization of the PDE and the boundary conditions. The computation of the components of B and y involves both the coefficients of the original PDE and the metric coefficients. In order to find out how the use of curvilinear structured grids affects the desired functionality of the framework we thus need to study how B and y are constructed.

3 Feature and variability modeling

Now, what features do we need in a PDE solver framework, in order to support the handling of curvilinear coordinates in a convenient way? Traditionally, in object-oriented modeling, there is no clear distinction between considerations about what features to support, and how to implement these features. Similarly, the variability of the framework is normally expressed implicitly, as a consequence of the implementation mechanisms used for the features, without discussion as to the desired *mode* of the variation, i.e., how and when the selection of features should take place.

Here, we take a different approach, which is inspired by the feature and variability modeling techniques that are by now common in the area of product line design [JGJ97]. Such modeling is appealing, since it makes explicit the scope of the framework, in terms of what features to support, and the desired variability of those features. The actual object-oriented modeling can then focus on how to best implement these features and variability.

3.1 Feature modeling

From the description given in Section 2, we identify a number of key concepts, which in the object-oriented modeling are seen as potential classes. Here we start out by considering them to be features (in boldface below), and examining any potential subfeatures (in italics below) that need to be supported, in order for the framework to handle curvilinear coordinates conveniently.

It should be noted that this selection makes explicit not only which features will be supported by the framework, but also which will *not*. Using an unstructured grid is thus not listed as a feature below, since we limit our analysis to PDE solvers using curvilinear structured grids.

Coordinate mappings

Each point in logical space can be described by d_l curvilinear coordinates. The corresponding position in physical space is described by d_{ph} Cartesian coordinates. For each of these there is a mapping

$$x_i = x_i(\xi_1, \ldots, \xi_{d_l}). \tag{1}$$

If the coordinate lines of the curvilinear coordinates intersect at right angles in physical space the coordinate mapping is *orthogonal*. The coordinate mapping is *non-orthogonal* if this is not the case.

The *metric data* that will be needed for formulating the PDE, including the boundary conditions, depends not only on the PDE itself, but also on the choice of coordinate mappings.

Grid

Here we discuss subfeatures of the logical grid together with those of its image, the physical grid.

The domain on which the PDE is defined can be described by curvilinear coordinates. The number of such coordinates needed gives the *dimensionality*, d_l, of the logical space.

The *dimensionality*, d_{ph}, of the physical space may be the same or larger than that of the logical space.

The grid is laid out along the curvilinear coordinates in logical space. The *dimensionality*, d_g, of the grid may be the same as, or smaller than d_l, the dimensionality of the logical space,

If the grid is laid out evenly in a coordinate direction, then it is *equidistant* in that direction, and the spacing, h, is found as

$$h_i = \frac{l_i}{n_i - 1}, \tag{2}$$

where l_i is the total length of the logical space in the direction of the i:th logical coordinate, and n_i is the number of points in that direction.

If the grid is *non-equidistant*, then it has different sized *grid spacings* in that direction.

The first and last point of the grid, in a certain coordinate direction, may be placed *on-boundary* or *boundary centered.*

There may also be certain topological factors which have bearing on the grid, such as e.g., *periodicity.*

Grid generator

The Cartesian coordinates of the grid points are given by a *grid generator.* Such a generator may also give the derivatives of the Cartesian coordinates with respect to the logical coordinates. The set of *metric data provided* by the grid generator is thus a subfeature.

Any calculation based on the metric data provided by the grid generator cannot be of higher *accuracy* than that of this data itself, which is therefore an important subfeature.

Representation of metric data on the grid

Different numerical methods have different requirements as to the *accuracy* of the metric data used. In some cases parts of the equations are discretized on grids which are *staggered* with respect to some *base grid*, and metric coefficients are thus needed at the grid points of those staggered grids as well.

One limit for what is possible to simulate in a PDE solver is given by the number of data points required for getting sufficient accuracy. It is important that data, that is not needed for the computation can be deleted in order to leave room for these large data sets. At what point in time allocation and deallocation of the storage needed for the metric data is thus an important subfeature, which we label *persistency.*

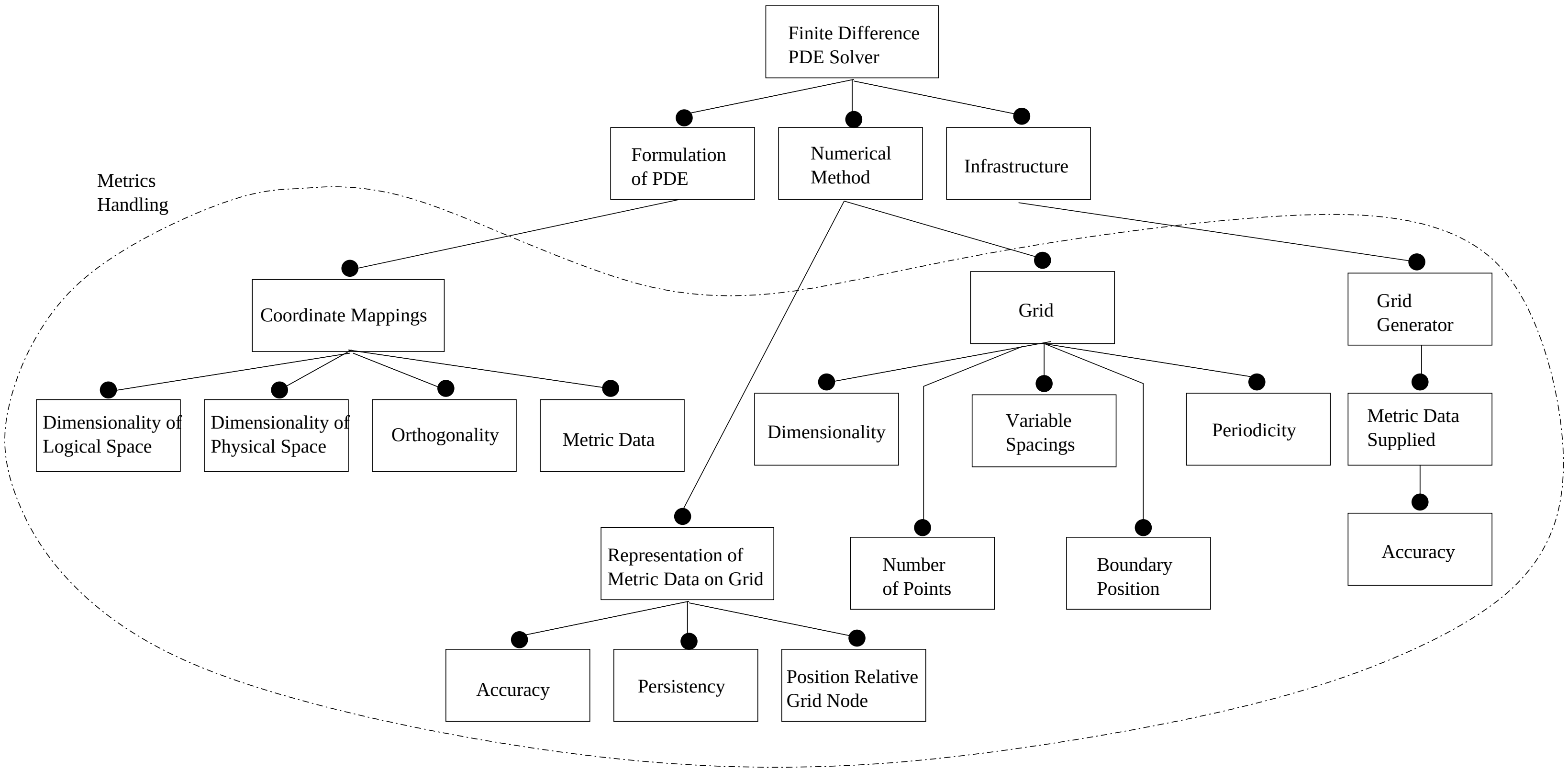

Figure 5: Feature diagram.

3.2 Feature diagrams

The result of the feature modeling carried out above is a list of features and subfeatures needed to support the required functionality.

The Coordinate Mappings is an important feature, and relevant subfeatures are the dimensionalities of the spaces they map between; the logical and the physical space, respectively. Whether or not the mappings are orthogonal, and which metric quantities are needed in order to formulate the PDE using the curvilinear coordinates are the other two subfeatures identified.

For the grid and its representations in logical and physical space a number of subfeatures are identified, The dimensionality of the grid can be smaller than or the same as that of the logical space. For each of the grid directions the number of points along that direction, and whether or not these points are equidistant must be determined. How the first and last grid point are placed, and whether or not there are periodicity conditions for these points are also relevant subfeatures.

The numerical method uses metric data on the grid. Relevant subfeatures here are the accuracy and persistency needed, and also at which positions relative to the grid note points that the metric data is needed.

The set of metric data provided (e.g. coordinate values, coordinate derivatives, etc.) and the accuracy of this data are consequences of the selection of a Grid Generator.

The feature diagram in Figure 5 shows the features and subfeatures listed above, and how these are related to more general features of the finite difference PDE solver we are modeling.

In this diagram all the leaves are *dimensions*, which means that they have a set of alternative subfeatures. For simplicity these have been left out of the diagram. Each of the other nodes relating to metrics handling also have a set of alternative subfeatures, which are not explicitly shown in the diagram. Together the nodes shown constitute the *variation points* for the metrics handling functionality. The notation that is used for the feature diagrams is described in more detail in [CE00].

3.3 Variability modeling

The next step of the variability modeling is to identify the supported options for the variation points. Variability modeling also includes deciding when and how selection of features should be performed, i.e., the binding times and binding modes [Bos00].

If features are bound at *compile time* we get a specific application. In the other extreme an application may include all possible features, and the selection is done during *run time*. Features can also be selected at *activation time*, i.e., when the program is started.

The binding mode of a feature indicates whether or not a feature selection can be changed. A *static* feature is bound once, and cannot be changed, whereas a *dynamic* feature is explicitly selected each time it is to be used. A *changeable* feature is bound, but can be rebound.

A module which is able to supply different features must have a *configuration interface*, through which a user or application programmer selects the relevant feature or features. This interface could be, for example, a compile pragma, command line option, or menu selection.

Depending on the binding time and binding mode of a feature, different implementation mechanisms can be used. Examples of such mechanisms are extension by linking in new code, configuration using parameters, and automatic code generation using a preprocessor. Other mechanisms which can be found in some object-oriented languages include template instantiation and inheritance through extension and/or interface specification.

Below we group the features with respect to implementation, i.e., features whose implementations are correlated are discussed together.

Dimensionalities

The grid, the logical space, and the physical space all have their own dimensionality. The relationship between the three is

$$d_g \leq d_l \leq d_{ph}$$

For each particular PDE solver d_{ph} has a fixed maximum value, which also applies to the other two dimensionalities. For the types of applications we are looking at, i.e., simulations of physical processes in ordinary space, d_{ph} will be limited to 2 or 3, but for other applications, such as for example molecular dynamics or financial simulations it may be much larger.

For performance reasons it is preferable to have the dimensionalities static and set at or before compile time. This allows for efficient traversal of data represented on the grid.

The different dimensionalities are implemented through the use of different underlying data types for representing data on the grid. Which data type to use depends on the desired dimensionality. In languages that support templates the dimensionality of these data types may be set as a template parameter c.f. [Ban00] [Vel98].

Grid

Since we expect each run of the PDE solver to be on fixed grids the number of points used in the discretization of each of the logical coordinates, n_i, should be statically selectable at activation time. Their values can be given as an input parameters.

Equidistant grids allow for more efficient computation of numerical approximations of derivatives, since the distance between consecutive points only needs to be computed once for all points along a grid line in logical space.

Variable grid spacings, on the other hand, are more flexible, and equidistant grids may also be represented as variable.

Variable grid spacings, can be implemented through the use of arrays which specify the values of the components of the curvilinear coordinates for each grid point. In this way the *on-boundary* or *boundary centered* end points can also be represented.

The positions of the grid points in logical coordinates are kept static and settable at activation time, since grids are not changed while in use. This implementation supports structured adaptive mesh refinement (SAMR) techniques, since the adaptiveness of these methods is through the introduction of new, finer grids, which are also static while in use. Some mechanism for supplying the desired variable spacings must be present. This could be either a data file, or an analytic expression.

Grid generator

The *coordinate mappings* are visible only through the data produced by the grid generator, which we consider to be an external module. Selection of the coordinate mappings is done through the selection of a grid generator. This selection thus also determines whether or not the mappings are *orthogonal*, and the *accuracy* of the *supplied data*.

We expect each PDE application to use only one grid generator, so this may be set statically at compile or link time.

The use of a common interface for the different grid generators will make it possible to switch from one to another simply by linking to a different class or library.

Metric data needed

Which metric data that is needed depends on the PDE to be solved, and is thus application specific. In spite of this it is desirable for the software to be able to produce a varied range of metric data, so as to minimize the difference in software between different applications. For the case of orthogonal curvilinear coordinates we need software that can generate the following metric data at the grid points:

- the components of the Cartesian coordinates, and
- the scale factors η_i, i.e., the square root of the diagonal elements of the metric tensor.

When non-orthogonal curvilinear coordinates are used the following metric data will be needed as well:

- the entries $\mathcal{G}_{i,j}$ of the metric tensor
- the determinant of the metric tensor

The selection of metric data to supply could be picked differently, since some metric quantities can be derived from others.

The software should be able to generate the metric data both on the base grid and on the staggered grids, since many PDEs are best discretized through the use of grids which are offset relative to each other.

In order to support the generation of different sets of metric data for the orthogonal and the non-orthogonal case we introduce alternative versions of the relevant data types. The selection is then done at link time, through the selection of the appropriate modules.

Depending on the numerical method it would be desirable to be able to specify the *accuracy* of the metric data. In practice this is given by the accuracy of the data supplied by the grid generator, and cannot be selected independently.

Construction and persistency

An adaptation of the *Builder* pattern described in [GHJV95] can be used to facilitate the production of different types of metric data. The class relationships used in this pattern are shown in Figure 6, with the names of the corresponding classes in our design given in parenthesis.

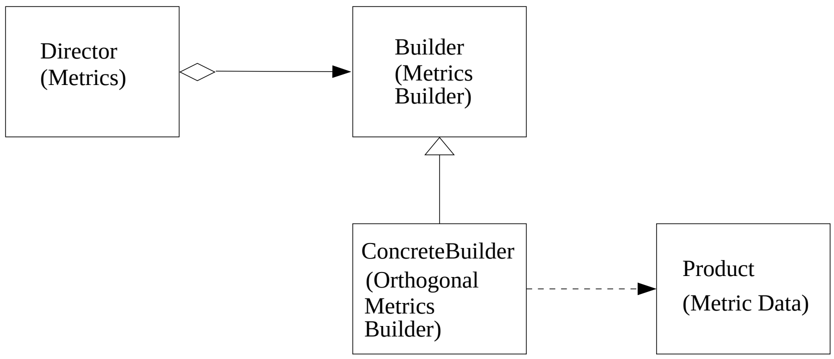

Figure 6: Object diagram for the builder pattern.

In the standard *Builder* pattern, the responsibility of the *Director* is to know which parts must be constructed. In our design, the Metrics class fills several roles. In addition to serving as the *Director* it also takes care of

- construction of the *Concrete Builder*, the class responsible for the production of the metric data,
- parameter parsing,
- construction of a representation of the grid in logical coordinates as well,
- construction and initializing coordinate mappings, in particular of the grid generator, and
- persistency of the metric data produced.

The *persistency* requirements are supported by providing methods, as well as default behavior, for

- generation of metric data,
- access to metric data, and
- deallocation of metric data.

4 Software model

Based on the concepts, features, and on the suggested implementations described in Section 3 a set of classes to support the necessary functionality are identified. Here we discuss these classes and their relationships.

An overview of the software model is shown in Figure 7, as a UML class diagram [RJB98] and will be discussed here. The Metric Builder is responsible for generation of the values at the grid points of the Cartesian coordinates and the metric tensor. The Orthogonal Metric Builder and the Non-Orthogonal Metric Builder are the *Concrete Builders* for the case of orthogonal and non-orthogonal coordinate mappings, respectively, and generate different representations of the metric tensor.

The positions of the grid points in logical space, i.e., in the curvilinear coordinates, are represented by the Logical Grid. The Logical Grid keeps track of the coordinate values in curvilinear coordinates for the grid points along each of the grid axes. The Metrics class, which acts as the *Director*, is responsible both for constructing the Logical Grid, and for giving it as an input to the Metric Builder to use in the production of the metric data. Selection of the number of points in the grid, n_i, the grid spacings, the placement of the end points, and the variable grid spacings, if these are used, are all given as parameters by the Metrics class when constructing the Logical Grid.

The mappings describing the relationship between the Logical Grid and the Metrics is represented by the Coordinate System class. The Coordinate System should be able to produce metric data for *any* Logical Grid.

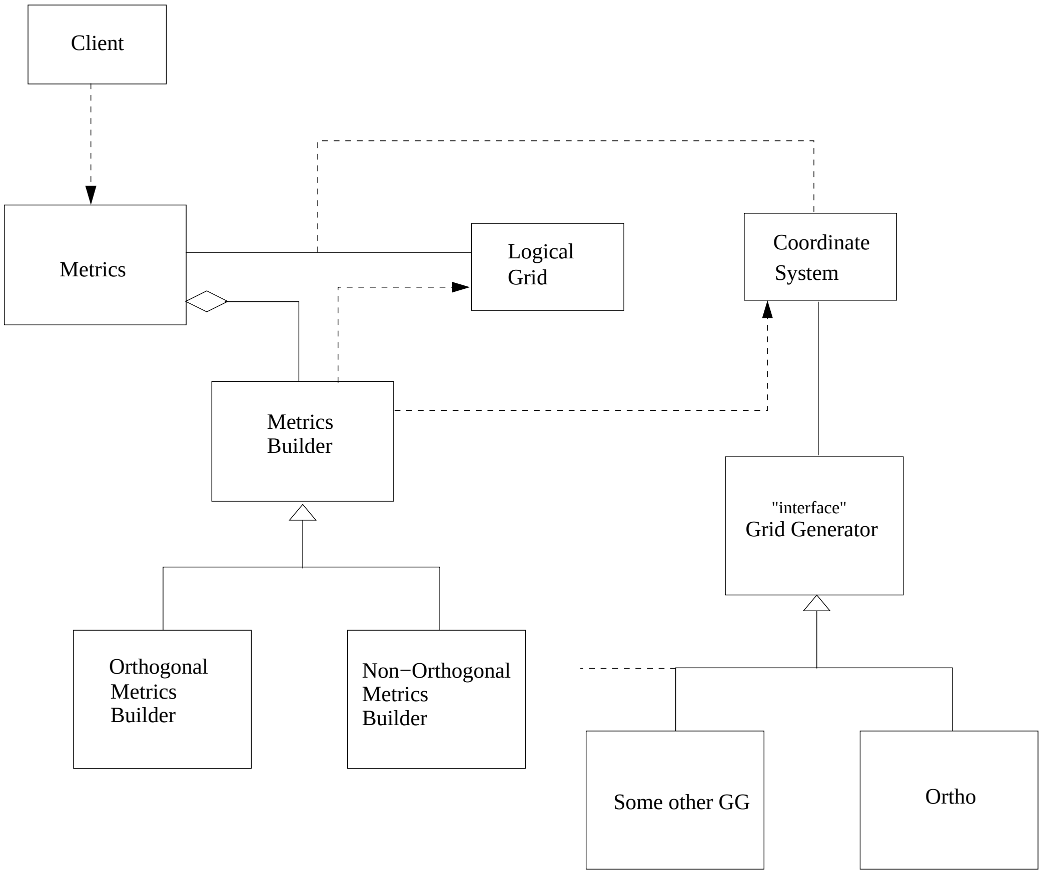

Figure 7: Class Diagram for the metrics handling classes.

The mechanism for this is the use of an associated Grid Generator, which is usually an external library routine. A common interface for the different Grid Generator classes makes it easy to switch between them. This interface provides the Cartesian coordinates and their derivatives with respect to the curvilinear coordinates at the grid points, as well as the accuracy with which these numbers are given. The interface follows the *Strategy* design pattern defined in [GHJV95]. For each new library that is to be used, a new version of the class must be implemented. The inheritance hierarchy for the Grid Generator classes is illustrated in Figure 7, showing the Ortho Grid Generator [Abr91] as an example.

The Metrics class is responsible on one hand for the creation of all other objects needed for the generation of metric data, and on the other hand for the generation and persistency of the metric data itself. When to generate what metric data, and how long this data should be kept is determined by the use of relevant access methods in the Metrics class.

The Client in Figure 7 represents any software module which uses the data supplied by the Metrics. This could typically be for the discretization of a PDE using some numerical method.

Remark: The above analysis is based on the idea that the domain can be described by *one* Grid. If we are dealing with a more complex domain, or if we want to split the domain for reasons of parallelization this may not suffice. In these cases we will need a collection of Grids, both Logical Grids, and Grid Fields, describing the metric data on these, together with a set of rules for how these connect.

Including these Composite Grids in the model is straightforward, and how these can be used to support parallelization through domain decomposition is discussed in [Lju03].

For the representation of data on the grid a family of Grid Field classes from the TENGO framework is used. These data can be scalars, vectors, or higher order tensors. In order to enable curvature dependent operations, e.g., the scalar product, on these, the TENGO Grid Field classes should be enhanced to include a connection to the underlying metric through the Metrics class, as shown in Figure 8.

The dimensionality of the grid, d_g, gives the array dimensions of the data, while the dimensionality of the physical space, d_{ph} gives the number of components of the vectors and tensors given on the grid.

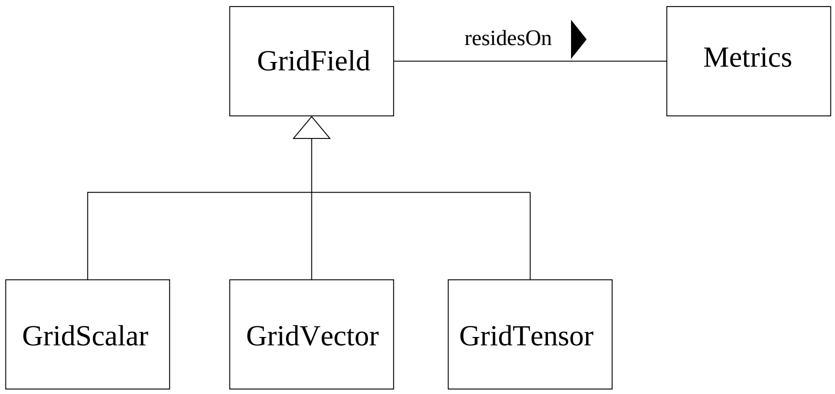

Figure 8: Class Diagram for the Grid Field classes.

5 Comparison of approaches to handling of curvilinear coordinates

In the accompanying article [LO04] the software model described above is validated against the usability design goals mentioned in Section 1. The model, together with a pilot implementation based on the model, is examined with respect to the usability concept, here given a contextual interpretation in terms of efficiency and flexibility.

The efficiency of the pilot implementation is comparable to, in parts even higher than, that of a pre-existing application-specific reference code. With regards to flexibility it is shown that the new software model performs well for a set of four change scenarios selected by an expert user group.

Here we present an alternative way of assessing the design, which is through comparison with other approaches to the issues of coordinate transformations and curvilinear coordinate systems, and how/whether other designs are able to provide the variability which we have discussed in Section 3.

Existing PDE solvers use various software representations to deal with the coordinate mappings. There are also packages which work on unstructured grids, but which make use of the simplified PDE formulations that the symmetries of a curvilinear coordinate system offers.

We will now group a number of existing software packages based on to what extent the software abstractions support operations on tensor fields, particularly the application of differential operators. Three main categories were identified.

1. In the first category there is no support at all for operators, or curved spaces. Packages belonging to this category are more concerned with efficient implementation on large, and often adaptive, grids. Examples of such packages are GrAL [Gra02], KeLP [KeL01], GrACE [GRA] and hypre [hyp01].

2. In the next category, illustrated in Figure 9, the application of differential operators is performed by some "external agent", such as for example a main program, or a "construct divergence" method, using classes which have no specified relationship to each other.

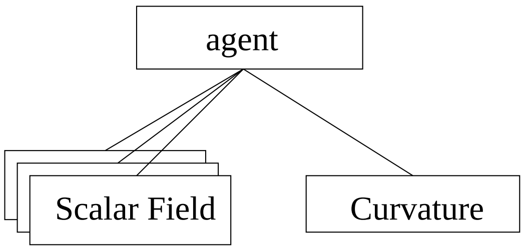

Figure 9: Object Structures for Category 2.

The Scalar Field represents the values of the components of the tensor on the grid. A simpler abstraction which does not have a partial differentiation operator defined may be used instead, which leaves the agent responsible also for this task. The Curvature contains geometry and curvature information. This may also be split up over several objects. As an example the coordinate values at the grid points can be stored in one object, while the corresponding values for the coordinate derivatives are stored in another.

POOMA [POO99], SAMRAI [Hor00] and PETSc [PET02] are packages whose software architectures belong to this category. In the POOMA [POO99] package, the only curvilinear coordinates which are supported are those which can be represented analytically. In the SAMRAI [Hor00] package there is a placeholder for the user to supply his or her own geometry. The Matrix-Free Matrices of PETSc [PET02] provide another way for the user to insert a curvature, by means of variable stencil coefficients which have to be specified.

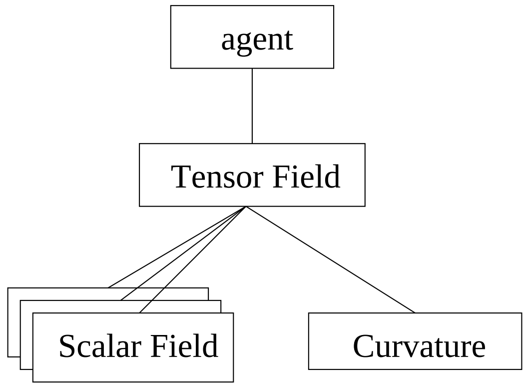

Figure 10: Object Structures for Category 3.

3. In the third category, illustrated Figure 10, the Tensor Field is responsible for constructing the numerical approximation of the differential operator. Knowledge about what numerical approximation to use, which grid to work on, and the curvature of the space are in this case intrinsic to the Tensor Field, through the Curvature object.

   Sophus [HFJ99], Cogito [Tea97] and Overture [Ove99] [Ove03] are frameworks which have a structure which corresponds to Category 3.

Sophus is well described by Figure 10, while in Cogito [Tea97] the coordinates and their derivatives are a property of the Field classes directly, instead of through some type of Curvature object.

In Overture [Ove99] [Ove03] the `MappedGridFunctions` know about the coordinate mapping, through the `MappedGrid`, which corresponds to the Curvature, as illustrated in Figure 11. A `MappedGrid` has attributes for the coordinates, the coordinate derivatives and for the Jacobian, i.e., the determinant of the metric tensor. The Tensor Field and its Scalar Fields are represented together in a `MappedGridFunction` object in Overture.

The variability requirements identified in Section 3.2 are not relevant for frameworks belonging to the first category, since they do no support curvilinear coordinate systems.

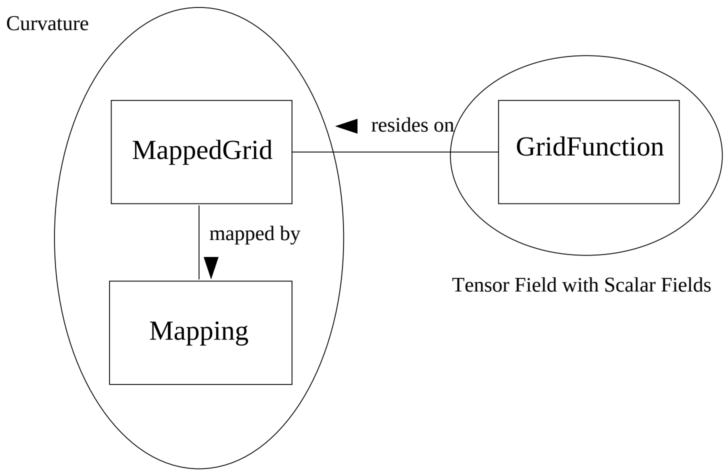

Figure 11: Overture Class Definitions interpreted in the light of Category 3.

In designs belonging to the second category the agent has to control any variation of the Scalar Field or the Curvature. Neither of these can be changed without influencing the agent in some way. Also the Scalar Field has an *implicit* dependence on the Curvature, which means that it is up to the agent to make sure that the geometry described by the Curvature actually matches that on which the Scalar Field is defined.

In the third category the dependence of the Scalar Field on the geometry has been made *explicit* by the coupling to an underlying Curvature. This means that the responsibility of matching a Scalar Field with its geometry has been removed from its user. The geometry can be varied without affecting the way the user formulates the PDE. The variability is thus best contained in frameworks belonging to the third category.

6 Conclusions

Setting our object model in relationship to those surveyed here, we note that its structure belongs to Category 3 (see Fig. 10 and 12). As in Overture the Scalar Field and the Tensor Field are treated together, and are using a common interface towards the Metrics. The Metrics, together with the Coordinate System, form a representation of the Curvature of the space, on which the Field is defined, and letting the Field know about the Curvature will free any user of the Field of having to keep track of this relationship.

The main result of the present article is the software model presented in Figure 7. This model was derived in a systematic way, to achieve the variability required by our *user group*. Moreover different alternatives for *implementing* this variability were considered. The arguments pro and counter these alternatives led us to the final model shown in Figure 7. Consequently,

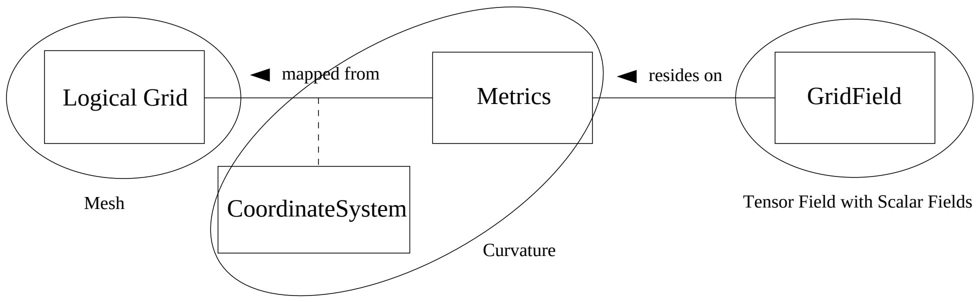

Figure 12: Object Structure for Our Model

the analysis carried out above provides a solid underpinning for the resulting model.

In conclusion, we claim to have proven, in an informal sense, the appropriateness of this model for the metrics handling subset of the kind of PDE solver framework under consideration. This claim gains further support by the observation made above that Sophus and Overture use similar models, however arrived at via different approaches.

Our goal is to extend the framework, so as to allow for increased flexibility with regard to the construction of numerical operators. When curvilinear coordinates are used, the numerical operators will include metric coefficients. This means that the software model developed here will also be an important part of the software model that will be developed for the flexible numerical operators.

References

[Abr91] L. Abrahamsson. Orthogonal grid generation for two-dimensional ducts. *Journal of Computational Applied Mathematics*, 34:305–314, 1991.

[Åhl99a] K. Åhlander. An extendable PDE solver with reusable components. In V. V. Kudriavtsev and C. R. Kleijn, editors, *Computational Technologies for fluid/thermal/structural/chemical systems with industrial applications*, volume 397-1, pages 39–46, New York, 1999. ASME.

[Åhl99b] K. Åhlander. *An object-oriented framework for PDE solvers*. PhD thesis, Uppsala University, Sweden, 1999. ISBN 91-554-4379-6.

[ÅO04] K. Åhlander and K. Otto. Software design for finite difference schemes based on index notation. *Future Generation Computer Systems*, 20, 2004. in press.

[Ban00] Wolfgang Bangerth. Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II. In Michel Deville and Robert Owens, editors, *Proceedings of the 16th IMACS World Congress 2000, Lausanne, Switzerland, 2000*, 2000. Document Sessions/118-1.

[BL97] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, 23:50–80, 1997.

[Bos00] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000. ISBN 0 201 67494 7.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[CZ01] S. Commend and T. Zimmermann. Object-oriented nonlinear finite element programming: A primer. *Advances in Engineering Software*, 32:611–628, 2001.

[DBQ99] W. Henshaw D. Brown and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In M. Henderson, C. Anderson, and S. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, 1999. SIAM.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[GRA] The Applied Software Systems Laboratory, Rutgers University. *GrACE Class Structures*. `http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/`.

[Gra02] G. Berti, Institut für Mathematik, Brandenburgische Technische Universität Cottbus. *The Concepts of the Grid Algorithms Library*, 0.2 edition, 2002. `http://www.math.tu-cottbus.de/~berti/gral/`.

[Hav00] M. Haveraaen. Case study on algebraic software methodologies for scientific computing. *Scientific Programming*, 8:261–273, 2000.

[HFJ99] M. Haveraaen, H. André Friis, and T. A. Johansen. Formal software engineering for computational modelling. *Nordic Journal of Computing*, 6:241–270, 1999.

[HK02] R. Hornung and S. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[Hor00] R. Hornung. Application development with the SAMRAI software framework. Technical Report UCRL-MI-133555 Rev. 1, Lawrence Livermore National Laboratory, P.O.Box 808, Livermore CA 94551-0808, USA, 2000.

[HT92] K. Högström and M. Thuné. Portability and data structures in scientific computing – object-oriented design of utility routines in Fortran. In W. Joosen and E. Milgrom, editors, *Parallel computing: from theory to sound practice*, pages 585–588, Amsterdam, 1992. IOS Press.

[HW99] R. Hornung and C. Woodward. An object-oriented approach for development and testing of parallel solution algorithms for nonlinear PDEs. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM, 1999.

[hyp01] Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL), Livermore, CA 94551, USA. hypre *high performance preconditioners, User's Manual*, 1.6.0 edition, July 2001.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse, Architecture, Process and Organization for Business Success*. ACM Press, 1997. ISBN 0-201-92476-5.

[KeL01] Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114, USA. *KeLP User Guide*, 1.4 edition, February 2001.

[KS94] P. Knupp and S. Steinberg. *Fundamentals of Grid Generation*. CRC Press, 1994. ISBN 0-8493-8987-9.

[Lju03] M. Ljungberg. Handling of curvilinear coordinates in a PDE solver framework. Master's thesis, Uppsala University, Department of Information Technology, 2003. IT Lic Series 2003-004.

[LO04] M. Ljungberg and K. Otto. Curvilinear coordinates in a PDE solver framework; Validation. Technical Report 2004-032, Uppsala University , Department of Scientific Computing, Box 337, 751 05 Uppsala, Sweden, 2004.

[LS99] K. Long and B. Van Straalen. PDESolve: An object-oriented PDE analysis environment. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM, 1999.

[MOT97] E. Mossberg, K. Otto, and M. Thuné. Object-oriented software tools for the construction of preconditioners. *Scientific Programming*, 6:285–295, 1997.

[ORT96] P. Olsson, J. Rantakokko, and M. Thuné. Software tools for parallel PDE on composite grids. In A. Ecer et al., editor, *Parallel Computational Fluid Dynamics—Implementation and Results Using Parallel Computers*, pages 725–732, Amsterdam, 1996. Elsevier Science B.V.

[Ove99] Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory (LLNL), Livermore, CA 94551, USA. *A Primer for Writing PDE Solvers with Overture*, 1.0 edition, October 1999.

[Ove03] Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory (LLNL), Livermore, CA 94551, USA. *Mappings for Overture, A Description of the Mapping Class and Documentation for Many Useful Mappings*, January 2003.

[PET02] Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA. *PETSc Users Manual*, anl-95/11 - revision 2.1.3 edition, May 2002.

[POO99] Los Alamos National Laboratory. *Parallel Object-Oriented Methods and Applications, POOMA Tutorial*, 2.1 edition, November 1999. `http://www.acl.lanl.gov/pooma/html/`.

[PS02] C. Pflaum and D. Seider. Efficient implementation of operators on semi-unstructured grids. In J. J. Dongarra P. M. A. Sloot, C. J. Kenneth Tan and A. G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 3. Springer, 2002.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998. ISBN 020130998X.

[TÅL+01] M. Thuné, K. Åhlander, M. Ljungberg, M. Nordén, K. Otto, and J. Rantakokko. Object-oriented modeling of parallel PDE solvers. In R. Boisvert and P. Tang, editors, *The architecture of Scientific Software.* IFIP, Kluwer Academic Publishers, 2001. ISBN 0 7923 7339 1.

[Tea97] M. Thuné et al. Object-oriented construction of parallel PDE solvers. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 203–226. Birkhäuser, 1997.

[Vel98] Todd L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[Yee66] K. Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation*, 14(3):302–307, 1966.