Towards an Expressive Language for PDE Solvers

Michael Thuné and Krister Åhlander*

Uppsala University, Dept. of Scientific Computing, P.O. Box 120, S-751 04 Uppsala, Sweden. E-mail: Michael.Thune@tdb.uu.se, Krister.Ahlander@tdb.uu.se

Abstract. Many significant real-life applications, e.g., air-craft design and environmental modelling, involve simulations based on the numerical solution of systems of time-dependent partial differential equations. In general, the computational domain has a complicated geometry, and the computations demand high-performance computers. The present paper lays a foundation for an expressive notation for this kind of application. By an object-oriented analysis concepts are identified, that can be combined flexibly enough to allow a programmer to express real computational problems (as opposed to model problems). The resulting concepts have been implemented as classes in C++. In a sense, this class library (referred to as Cogito/Solver) will specialize C++ into a language for the application domain under consideration. However, the aim is to use Cogito/Solver as a basis for higher-level interfaces. As a first attempt in this direction, an object-oriented database for Cogito/Solver objects has been implemented. Through an interface to this database, a programmer can compose an executable object (a *Numerical Experiment*), from various smaller objects, related to the different Cogito/Solver concepts.

Keywords: scientific computing, object-oriented, parallel, composite grids

1 Introduction

The two main demands on a programming language are: *expressiveness* (of the source text) and *efficiency* (of the executable code). In scientific computing the latter is considered so important that Fortran remains the dominating language in the area, despite its low level of abstraction. The challenge is to design a new alternative, considerably more expressive than Fortran, but with comparable efficiency.

This can be regarded as the long-term goal of the Cogito project [14]. The focus is on an important subfield of scientific computing: the numerical solution of systems of time-dependent partial differential equations. Such systems arise in many significant real-life applications, e.g., air-craft design and environmental modelling. In general, the computational domain has a complicated geometry

* The work presented here was supported by a grant from TFR, the Swedish Research Council for Engineering Sciences.

(e.g., a complete air-craft [4]), and the computations demand high-performance computers.

In this paper we will lay a foundation for an expressive notation for this kind of application. By an object-oriented analysis of the problem area in question, we identify concepts that can be combined flexibly enough to allow a programmer to express real computational problems (as opposed to model problems).

The resulting concepts have been implemented as classes in C++. In a sense, this class library (referred to as Cogito/Solver) will specialize C++ into a language for our field of applications. However, we aim at using Cogito/Solver as a basis for higher-level interfaces. As a first attempt in this direction, we have implemented an object-oriented database for Cogito/Solver objects. Through an interface to this database, a programmer can compose an executable object (a *Numerical Experiment*), from various smaller objects, related to the different Cogito/Solver concepts.

The present implementation is for problems in one space dimension. However, the object-oriented analysis and design have been pursued for general problems, and an implementation for problems in several space dimensions is being planned.

What about efficiency? The Cogito project started with a consideration of efficiency issues. For this reason the Cogito software tools have been structured in three layers, with Cogito/Solver on the highest level of abstraction. The mid layer, Cogito/Grid, provides classes (implemented in Fortran) for so called *composite* computational grids (relevant for treating complicated geometries, in the context of finite difference methods), and corresponding grid functions (for the representation of data on composite grids). Cogito/Grid uses an SPMD approach to parallel programming, and code written on this level can be executed on a large number of platforms, serial, and parallel of MIMD type with distributed memory. The basis for this is the lowest level, Cogito/Parallel. It contains classes (implemented in Fortran) for message passing, data distribution, etc. Our results so far indicate that programs based on the two lower Cogito levels are comparable to plain Fortran code with respect to execution time and parallel sizeup [9].

The intention is that the next implementation of Cogito/Solver will be based on Cogito/Grid, which should make the resulting programs efficient. This will tie together all parts of the Cogito project, making it possible to realize an expressive *and* efficient alternative to Fortran.

Most related work is very recent. Williams [15] proposes an object-oriented approach to unstructured-grid problems, in particular finite element methods. He adopts the view that his classes specialize C++ into a special purpose language, DIME++, for his class of applications.

Other researchers, though not explicitly discussing language issues, aim at raising the level of abstraction in scientific computing. Lemke and Quinlan [8] describe a class library intended to be a basis for implementations of structured grids. Karpovich et al. [7] discuss an object-oriented approach to stencil algorithms. Compared to these efforts, we aim at a higher level of abstraction.

Lately, a research group at Los Alamos has undertaken an effort to implement higher-level classes on top of Quinlan's A++ [5]. Their work is closely related to our classes on the Cogito/Grid level.

The work by Gropp and Smith [3] is similar to ours in spirit, but with focus mainly on tools for parallel iterative solution of linear systems.

The POOMA framework [10] addresses the same general issues as Cogito, and with a similar approach, but with differences in details. They focus on scientific computing in general, whereas we restrict our attention to a specific (and important) class of problems. Moreover, the POOMA framwork has five layers, while Cogito has three. Our bottom two layers seem to correspond to the bottom four layers of POOMA. Finally, the layer discussed in the present paper, i.e. Cogito/Solver, has no apparent counterpart in POOMA.

Fritzson et al. [2] have an interesting approach, combining symbolic algebraic manipulations and numerical computations in the framework of an object-oriented environment, ObjectMath. So far, their emphasis has been on mathematical modelling. Recently, a cooperation has been established between the ObjectMath and Cogito groups.

2 Expressiveness

Before presenting the details of our work, it is appropriate to give a more precise formulation of the task we set out to solve. We consider a particular application area, for which we aim at designing an expressive special purpose programming language (or, more generally, a problem solving environment). What do we mean by an "expressive" programming language? An expressive language will allow an application specialist to express computational problems in the application area

- flexibly (e.g., not restricted to predefined numerical methods or mathematical models)
- generally (i.e., not restricted to model problems)
- briefly (this is a condition where Fortran fails)
- with concepts that belong to the application area, thus making the program text immediately meaningful to an application specialist (this is another condition where traditional Fortran programs fail[2]).

This is not meant to be a *definition* of expressiveness. Rather, the intention is to point out some aspects of the concept, that we find essential.

A further aspect of importance in our context is that an expressive description should be easy to modify, *without* complete reformulation. For example, changing or modifying the numerical method should only amount to a small and local modification of the program. This is *not* the case in traditional Fortran programs for our kind of application. Typically (and strongly simplified) the innermost loop of such a program has the following form:

```
loop over all grid points (i, j, k)
```

[2] The situation is improved by Fortran 90, with its support for data abstraction.

```
    $v_{ijk}^{n+1}$ := right-hand-side
end loop
```

Here, v_{ijk}^{n+1} is the solution (in general an array of values) to be computed at the grid point with indices ijk, and for discrete time t_{n+1}. The formula for **right-hand-side** involves (a) coefficients coming from the PDE problem and (b) difference operators applied to (c) grid functions. In general, the coefficients may depend on (d) grid coordinates, (e) time, and (f) grid function values. The different components (a)–(f) are inseparably mixed in the loop-body. Thus, changing the numerical method will amount to modifying large parts of the program. Similarly, for changing the equations a major reformulation of the code will be necessary.

To summarize: An expressive programming language for our kind of application should be based on concepts from this application area. However, in the description of the complete computational problem, these concepts tend to be inseparably linked to each other, and consequently the resulting programs are difficult to modify.

For this reason, our approach to finding an expressive language is to analyze the application area from a conceptual point of view, i.e., to make an object-oriented analysis. Here, the goal is to find classes (concepts) such that all computations in the application domain can be expressed as interactions between instances of these classes.

3 Analysis

Presenting a complete object-oriented analysis is beyond the scope of this paper.[3] Our aim is rather to comment on parts of the analysis, in order to motivate the object model presented at the end of this section.

The object-oriented analysis begins with an identification of central concepts in the application domain (in our case: numerical solution of time-dependent partial differential equations). As a typical example, consider the simulation of airflow through an expansion pipe. Figure 1 shows simulated flow in a two-dimensional cross-section of the pipe. There are four plates in the pipe.

The airflow is described by a set of partial differential equations (*PDE*). They are valid in the interior of the pipe. At the inlet, outlet, at the plates, and at the pipe walls, ***boundary conditions*** must be prescribed. Finally, ***initial conditions*** are needed, describing the airflow at the beginning of the simulation.

The actual simulation is carried out by solving the mathematical problem numerically. We consider finite difference methods. Such methods are commonly used for the kind of applications we discuss here. In this context, the solution is approximated at discrete points, laid out so as to form a rectangular, structured *grid*. A complicated geometry can not be covered by a single rectangular grid.

[3] We have mainly used the OMT approach [11]. Moreover, we have considered use cases, according to Jacobsson [6]. Finally, our class diagrams have been drawn with the Rose tool, which supports Booch's method [1].

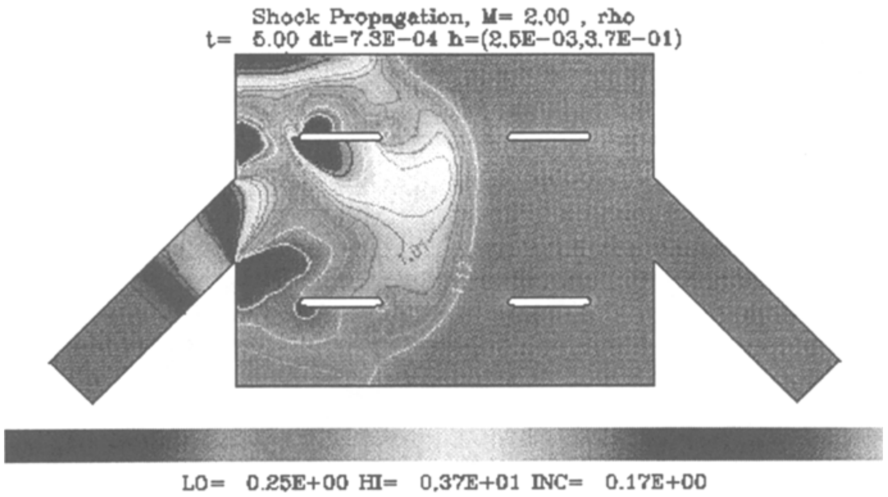

Fig. 1. Simulation of airflow through an expansion pipe

Consequently, it is common to use a *composite grid*, being a union of simple grids (see Figure 2, showing a composite grid covering the pipe in our example). The approximate solutions are represented as *grid functions*. A grid function takes on values only at the grid points.

The process of solving a time-dependent partial differential equation can be summarized as follows: Knowing the state of the unknown(s) on the first time level, the wish is to obtain the state of the unknown(s) T seconds later. A numerical method reaches this goal by discretization in space and time. The derivatives are approximated by discrete so called *difference operators*, both in space and time. Referring to the choice of such operators, we will use the terms *space discretization* and *time discretization*, respectively. First, we can discretize in space, handling space derivatives, and obtaining the right hand side in

$$\frac{d}{dt}\texttt{u} := \texttt{discreteSpaceOperator(u)} \tag{1}$$

This is an ordinary differential equation, to which a time discretization is subsequently applied.

To summarize the discussion so far, we have identified the following concepts from the application area: *Partial differential equation*, *Boundary condition*, *Initial condition*, *Grid*, *Composite grid*, *Grid function*, *Difference operator*, *Space discretization*, *Time discretization*. These are candidates for being regarded as classes in our application domain.

The continued analysis focusses on the dynamic interaction between objects of these classes. We consider first some informal use cases. They are expressed in

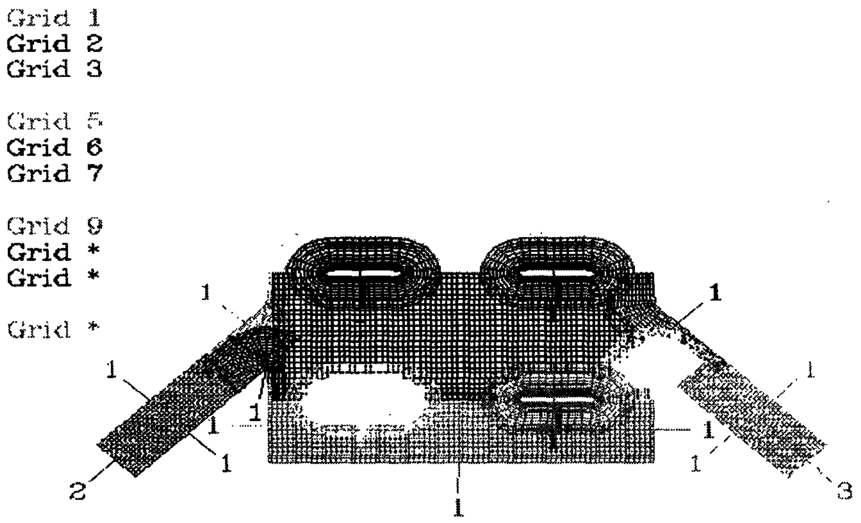

Fig. 2. The composite grid used in the simulation depicted in Figure 1

terms of users interacting with a "system". The system may be a programming language, having our classes as built-in data types. Another alternative is that the system is a problem solving environment, see § 4.

We distinguish between two main user categories:

- The problem oriented user who wants to solve some problem without caring about the numerical method.
- The method oriented user who wants to develop a numerical method that can be applied to different problems.

A typical use case initiated by a problem oriented user is sketched below:

Use Case 1 *Solve a previously defined problem with the aid of previously defined space and time discretizations*

The problem oriented user chooses a problem (including boundary conditions and initial condition), a grid, a space discretization and a time discretization. He or she decides which time step to use, and instructs the system to calculate the solution at time T seconds, storing intermediate results every, say, ten seconds.

This use case is "ideal" in the sense that no extra low-level programming needs to be done by the user. The ability to use the system without having to write low-level code is desirable to many problem-oriented users. It should for example be possible to experiment with different boundary conditions or time discretizations without the extra time for coding, compiling and verification of the code. Still, it is difficult to foresee every kind of PDE problem when constructing the system. The next use case emphasizes that a problem oriented user must be able to extend the system.

Use Case 2 *Solve a new kind of problem with the aid of previously defined discretizations*
The user realizes that the problem he or she is interested in is not yet available as a predefined data type. He or she extends the system by implementing the problem so that it fits within the system's framework, and then—after compiling and linking—proceeds as in the use case above.

The extendibility of the system is important for the method oriented user as well. The following use case would be typical for the method oriented user. Here, the system will be an excellent tool in the verification process, since a number of test cases would be found in the problem library.

Use Case 3 *Develop and verify a new kind of space discretization*
The method oriented user develops a new kind of, say, space discretization. He or she extends the system with this feature, taking advantage of components that the system offers, such as components for handling the boundary conditions. After compiling and linking, the user verifies the new method using an existing time discretization and a number of test problems with known solutions.

In view of these use cases, we point at a number of issues to be considered in the continued analysis.

1. The reusability of different components is important, especially in the second and third use cases, when the system is extended. In this context, reusability implies that the definition of a new component (not previously in the library; for example: a new PDE problem) can be achieved by writing only a few lines of low-level (e.g., plain C++) code, describing the new features of the component as compared to its ancestors in the inheritance hierarchy.
2. Numerical methods for PDE problems assume that the equations are stated in a suitable form. Basically, there are two kinds of formulations of the PDE: conservative or non-conservative. Conservative equations express conservation of a quantity and are of importance in physics. Although a conservative equation can always be restated in non-conservative form, some numerical methods are developped especially for problems on conservative form. How shall the description of the different kinds of problems and methods be structured?
3. There are different kinds of boundary conditions which need different treatment. Sometimes the handling of the given boundary condition affects not only the system's static structure, but the complete algorithm. Also, it is common that the given boundary conditions taken from the mathematical model are not sufficient for getting a stable method. So called numerical

boundary conditions must be added to the equations and consequently handled by the system in a convenient way.

4. Sometimes is it desirable to use one space discretization in one part of the domain and another space discretization in another part. How shall this be achieved?
5. As indicated in all the use cases, it must be easy to connect the different components. In particular, it should be straightforward to exchange one component (for example a boundary condition) without changing any of the other components.
6. Use Case 3 emphasizes the question of how different numerical experiments shall be compared. The system must allow for a flexible description of how and when to store intermediate results for subsequent post-processing.

In order to address these and similar questions, we propose the following object model (see Figure 3). The key classes are: Grid, Grid Function, PDE Problem, Space Handler, Time Handler, and Numerical Experiment. Note that difference operators are not considered as classes. In our model, they occur as operations on class Grid Function. This class is not shown in Figure 3, since it is not visible to the user to the same extent as the other key classes.

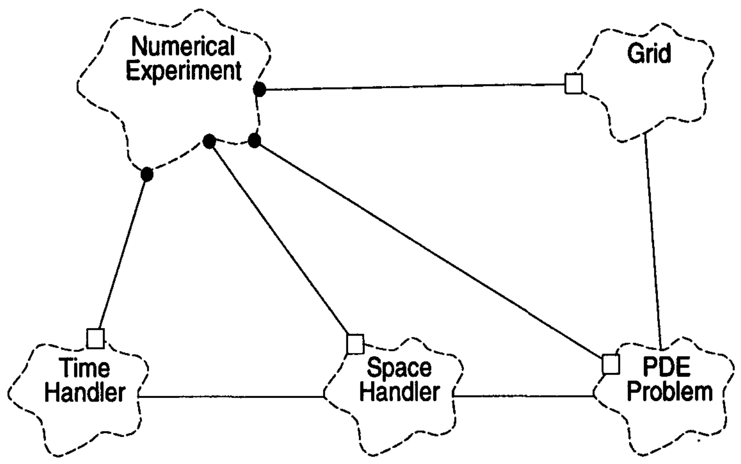

Fig. 3. Key classes of the proposed object model. Notation according to Booch. Solid lines denote associations between classes. Solid balls symbolize aggregates.

Grid and Grid Function are relatively transparent to the user of Cogito/Solver. However, both classes are key components with which to build new parts of the system (Issue 1).

PDE Problem is a composite class, consisting of Boundary Conditions and Initial Condition, and responsible for describing the equations. Since the system must support both conservative and non-conservative problems (Issue 2), two heirs are introduced (Figure 4). These classes are abstract base classes, so concrete problems will be heirs to either Non Conservative Problem or Conservative Problem. Each heir defines only the partial differential equation. In this way, the object model yields an extendible design. The Space Handler (Figure 5) is also

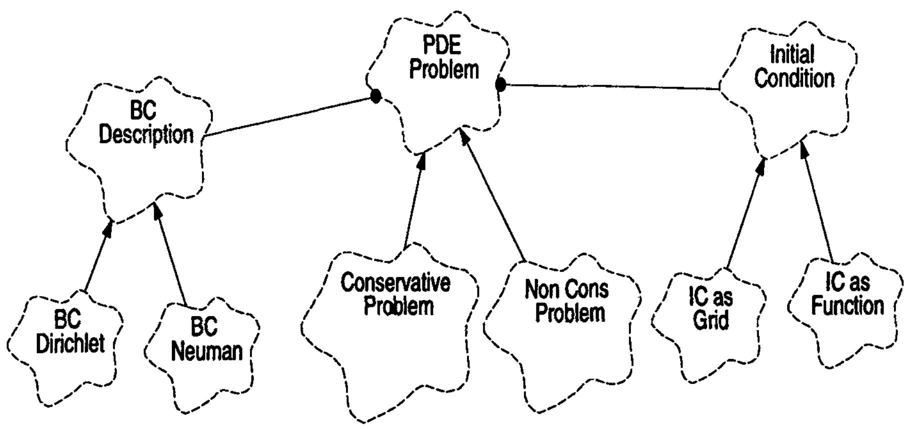

Fig. 4. Class PDE Problem with heirs. Notation according to Booch. Arrows denote inheritance.

a composite object, consisting of Space Discretizations and Boundary Handlers. The Boundary Handlers are components whose purpose is to handle Boundary Conditions (Issue 3). To address Issue 4 above, the Space Handler object may contain several space discretizations. The class Space Discretization is an abstract base class, and its heirs are Conservative and Non Conservative Space Discretizations. Their heirs are concrete classes representing the different space discretizations.

A similar discussion as for the Space Handler motivates the introduction of the class Time Handler, consisting of one or several Time Discretizations. Time Discretization is an abstract base class, with concrete inheritors such as RungeKutta. (The distinction between conservative and non-conservative problem is transparent to the Time Discretization classes. No extra heirs need to be introduced.)

The Numerical Experiment can be looked upon as being a control object. Its key responsibilities are to connect the correct objects in the correct way (Issue 5), and to control the outputs from the experiment. Thus, it provides means for comparing different experiments (Issue 6).

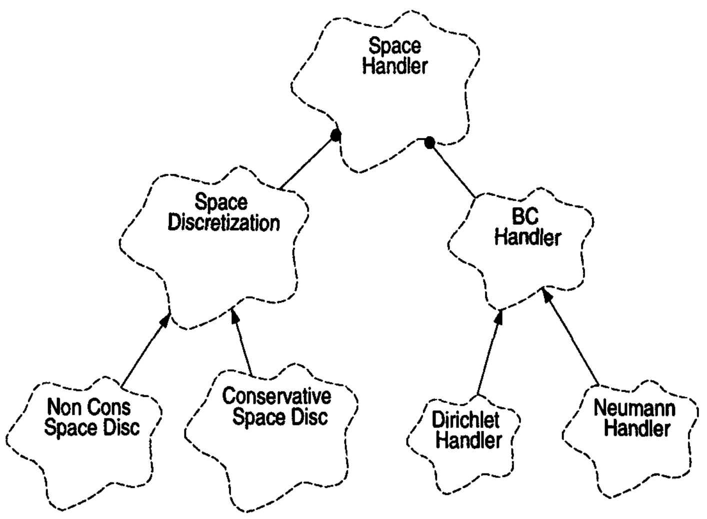

Fig. 5. Class Space Handler as an aggregate. Notation according to Booch.

The inheritance structure in the proposed object model makes it straightforward to extend the system with new solution methods, PDE problems, etc. In the scenario below, we demonstrate that the model in addition decouples the components, so that, e.g., the time handler can be replaced, while all other components are kept unchanged. This makes program modification easy. The key is an appropriate division of responsibilities between the classes, as is high-lighted in the scenario.

Scenario *The calculation of the solution on the next store-time level.*

The Numerical Experiment (i.e. an object of the Numerical Experiment class), asks the Time Handler to advance until the next store-time level. The order propagates to the Time Discretization objects. Each Time Discretization will advance one step at a time, requiring the Space Handler to calculate the right hand side of the semi-discretized equation (1). The process of calculating the right hand side will depend on the mathematical formulation of the PDE Problem, and on the chosen Space Discretizations. The Space Handler asks its Space Discretization objects to compute discrete derivatives, and calls upon the PDE problem for supplying the problem dependent information. When the right hand side has been calculated the control is returned to the Time Discretization.

The boundary conditions shall also be applied, and the Time Handler is responsible for *when* this should happen, since the order of doing this depends on the time discretization method. But the responsibility for *how* this should be done lies with the Space Handler, since these calculations depend on the space discretization method. The Space Handler delegates the task to the Boundary Handlers, which get the necessary information from Boundary Condition objects.

When the Time Handler has reached the store-time level, it returns the control to the Numerical Experiment which stores the solution, for subsequent post-processing.

4 Towards a Special Purpose Programming Language

To see the consequences of the results above, consider the situation in computational fluid dynamics, where the Navier–Stokes equations constitute an important mathematical model. Typically, CFD specialists say that they have a "Navier–Stokes solver". By this, they mean a huge Fortran program which contains the mathematical model, boundary conditions, grids, grid functions, and specifications of the numerical method to be used. The expression "Navier–Stokes solver" reflects that all the components of the program are inseparably linked to each other, making it difficult to replace one component without rewriting major parts of the code. Moreover, the initial construction of such a program is a significant task.

Within the framework of Cogito/Solver, the Navier–Stokes equations in conservative form would be a subclass of Conservative Problem. To create a new "Navier–Stokes solver", the programmer would simply create a new object of this class, and connect it to other objects, e.g., boundary condition descriptors, space handlers, etc. Given a library of predefined classes, a ready-to-run program could easily be composed. Subsequent modification would be equivalent to replacing some object by another object of the same class.

From this point of view, Cogito/Solver provides C++ with data types (classes) specializing it into a language for our particular class of applications.

As an example, we show the following program for the solution of a one-dimensional PDE problem similar to the (two-dimensional) Shallow-Water equations, which is a common model for flow in shallow water. The example is executable, using the present pilot implementation of Cogito/Solver.

```
int main()
{
  // Declare the main objects.

  NumExp myExp;
  NonConsProblem *problem = new ShallowWater(); // An heir to NonConsProblem
  SpaceHandler theSH;
  TimeHandler  timehandler;
  const int N = 100;          // Grid point indices from 0 to N
  Grid grid1(0.0, 1.0, 0, N); // This is a simple class Grid used for
                              // the pilot implementation.

  // Prepare the problem.
  // "ShallowWater_*" are previously declared functions, used to
  // initate initial conditions and boundary conditions.

  FcnInitCond ic( ShallowWater_IC);
  BCDirichlet bcleft(0,1,ShallowWater_BC);  // Same BC function
  BCDirichlet bcright(1,1,ShallowWater_BC); // at both boundaries
  problem->use( &ic);
  problem->add( &bcleft);
  problem->add( &bcright);

  // Set up the space handler.

  SpaceDisc * interiorSD = new Dzero;
  SpaceDisc * boundarySD1 = new Dplus;
  SpaceDisc * boundarySD2 = new Dminus;
  DirichletHandler bchd;
  NeumannHandler bchn;

  theSH.add( interiorSD );
  theSH.add( boundarySD1 );
  theSH.add( boundarySD2 );
  theSH.add( &bchd );
  theSH.add( &bchn );

  // Set up the time handler.

  double dt = 0.4/N ; // Set time step.
  TimeDisc * timedisc = new RungeKutta(dt);
  timehandler.add( timedisc );

  // Prepare the experiment.

  myExp.use( &grid);
  myExp.use( problem);
  myExp.use( &theSH);
  myExp.use( &timehandler);

  // Run until time 1.0, and store solution on file!

  myExp.solveUntil(1.0,"exShallow.m");

  return 0;
}
```

This new language is expressive from all the aspects discussed in § 2. However, the user who wants the flexibility, e.g., to use a boundary condition, which is not in the library, would have to do some coding in plain C++, in order to define a new subclass. This is not a completely satisfactory solution.

Consequently, we aim at raising the level of abstraction further. What we have in mind is a problem solving environment rather than a language. This environment should contain:

1. High-level interfaces for specifying concrete inheritors to abstract classes. (This way, defining, e.g., a new boundary condition will be straightforward.) These interfaces will assumedly have different forms for different classes. As an example, there could be a special interface for describing difference operators, and another interface for describing differential equations. These various interfaces would generate code for the new C++ subclasses.
2. An interface for creating new instances of existing classes, for storing such instances in an object data base, and for combining stored objects into complete numerical experiments.

The second goal has in principle been accomplished. The objects on the Cogito/Solver level were made persistent using the object data base TPS, Texas Persistent Store [12]. A menu driven interface called Cogito WorkBench, was implemented on top of the Cogito/Solver layer, and the user can combine and execute different objects flexibly [13]. He or she can choose between combining existing classes in new ways, and can also create new objects. Use Case 1 from the previous section is completely supported by Cogito Workbench.

In order to change boundary conditions at run-time, the possibility of interpreting simple C++ functions with one argument (time) was introduced. Even though these interpreting functions do not slow down the execution time significantly, it is also possible to generate C++ code from these functions. In this way, a step towards the first goal was taken.

5 Conclusions

We have proposed an object model for applications involving numerical solution of time-dependent partial differential equations. The classes of the object model, and the division of responsibilities between them, form a foundation for an expressive programming language for this application domain. We have implemented the object model in C++. The resulting class library, Cogito/Solver, will specialize C++ into a language for our field of applications. However, we aim at a higher level of abstraction, having in mind a problem solving environment rather than a language. Cogito Workbench (and the related object-oriented data base) as described in the previous section, is a first step towards this goal.

Expressiveness is not sufficient. In scientific computing, the efficiency of the executable code is crucial. To this end, we plan to reimplement Cogito/Solver on the basis of Cogito/Grid, which is a layer of classes (implemented in Fortran) supporting an SPMD approach to parallel programming of finite difference methods on composite grids. Cogito/Grid and its underlying layer Cogito/Parallel have been designed with efficiency issues in focus.

The aim is to continue work along two lines. One is to to develop the problem solving environment further. The second is to generalize Cogito/Solver, which in the present pilot implementation only handles problems in one space dimension. The object-oriented design has been pursued for multidimensional problems, and the next implementation will include these features. Moreover, the proposed design is intended to be adequate for composite grids. The structure of the classes will remain the same, and the changes will be local to the PDE Problem related classes, the Space Handler related classes, and of course the Grid and the Grid Function. Since the lower level layers of Cogito are already able to handle multi-dimensional problems and composite grids, the new implementation of Cogito/Solver, based on the lower layers, should be straightforward.

Acknowledgement

We thank Ola Strandberg for his work on Cogito Workbench.

References

1. G. Booch, Object-Oriented Analysis and Design with Applications, Benjamin/Cummings, 1994. Object-Oriented Analysis and Design with Applications,
2. P. Fritzson et al., *Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis.* In Proceedings of the 7:th International Conference on Technology of Object-Oriented Languages and Systems: TOOLS EUROPE'92, Prentice-Hall.
3. W. Gropp, B. Smith, *Scalable, extensible, and portable numerical libraries.* In Proceedings of the Scalable Parallel Libraries Conference, pp. 87–93, IEEE, 1994.
4. J. Häuser et al., *Parallel computing in aerospace using multi-block grids. Part 1: Applications to grid generation.* Concurrency, 4 (1992), pp. 357-376.
5. W. D. Henshaw et al., Private communication.
6. I. Jacobsson et al., Object-Oriented Software Engineering—A Use Case Driven Approach, Addison-Wesley, 1994.
7. J. F. Karpovich et al., *A parallel object-oriented framework for stencil algorithms.* In Proceedings of the Second International Symposium on High-Performance Distributed Computing, pp. 34-41, 1993.
8. M. Lemke, D. Quinlan, *P++, a parallel C++ array class library for archi-tecture-independent development of structured grid applications.* ACM SIGPLAN Notes, 28 (1993), pp. 21-23.
9. J. Rantakokko, *Object-oriented software tools for composite-grid methods on parallel computers.* Report 165, Dept. of Scientific Computing, Uppsala University, Uppsala, 1995.
10. J. V. W. Reynders et al., *POOMA: a framework for scientific simulation on parallel architectures.*
    Available on Internet, http://www.acl.lanl.gov/PoomaFramework/
11. J. Rumbaugh et al., Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.
12. V. Singhal, S. Kakkad, P. R. Wilson, *Texas: An efficient, portable persisitent store.* Dept. of Computer Sciences, Univ. Texas at Austin, Austin, Texas.
13. O. Strandberg, *Persistent objects in Cogito.* Internal Report No. 95-11, Dept. of Scientific Computing, Uppsala University, Uppsala, 1995.
14. M. Thuné, *Object-oriented software tools for parallel PDE solvers.* Invited paper at the International Conference on Parallel Algorithms (ICPA '95), Wuhan University, October 16–19, 1995.
15. R. D. Williams, *DIME++: A language for parallel PDE solvers.* Report CCSF-29-92, CCSF, Caltech, Pasadena, 1993.