AN OBJECT-ORIENTED APPROACH TO THE DESIGN OF FLUID MECHANICS SOFTWARE

Christophe Calvin[1], Olga Cueto[1] and Philippe Emonot[1]

Abstract. This article presents the guiding principles of the architecture of Trio_U, a new generation of software for thermohydraulic calculations. Trio_U is designed to serve as a thermohydraulic development platform. Its basic conception is object-oriented and it is written in C++. The article demonstrates how this type of design enables an open, modular software architecture.

1. Introduction

Trio_U is a thermohydraulic software tool that has been under development at CEA, since 1993. This article presents the guiding principles underlying the design of this software. It starts with a reminder of its overall objectives and a justification of the main choice made:

- object-oriented design;
- C++ language;
- thermohydraulic framework.

We will show how this enabled us to develop an open, modular software architecture.

1.1. The decision to choose object-oriented design

The code is supposed to deal with various thermohydraulic problems including turbulent one phase incompressible flows, two-phase compressible flows using various numerical methods including finite difference, finite volume and finite element methods. It is also supposed to be performant on a wide variety of computers including massively parallel computers. Abstraction and reusability is therefore a key issue. It should be possible (when it makes sense) to write very generic algorithms for a class of problems without explicit (unneeded) reference to a given discretization method (in time or space), or a specific physical model.

We want:

- an open software architecture to which new numerical schemes and physical models can easily be added;
- a modular structure from which sub-assemblies can easily be extracted to produce industrial software dedicated to specific applications;

---

[1] Commissariat à l'Énergie Atomique, 17 rue des Martyrs 38054 Grenoble Cedex 9, France. e-mail: `philippe.emonot@cea.fr`

- the platform to be suitable for hardware architectures comprising single or multiple processors. Parallel processing must be handled in such a way that when new physical models or numerical schemes are added there is no need to significantly change the software as a whole.

Object-oriented design would appear to be extremely suitable for producing an open, modular software architecture, with components that can be reused. In addition, implementing parallel processing must be transparent for developers who are either specialists in computing or fluid mechanics and who will be the "clients" of the software platform. This article will try to show why this appears to be the case.

1.2. Merits of object-oriented design and programming

Why did we opt for "object-oriented design" rather than methods based on a functional decomposition? One answer, which is of prime importance, is that with functional decomposition there is often insufficient data abstraction. We therefore concluded that design based on functional decomposition would be:

- less capable of adapting to changes,
- less suitable for developing a modular software architecture (in the sense that it consists of software modules that may be used separately).

The problem is that using functional decomposition means that interesting data lose their specificity, for when a system is organized in the form of a function tree, any data accessed by two functions must be global for both. As a result, an interesting piece of data rises further towards the top of the tree as more and more functions asks to access it. In contrast, object-oriented design tends to concentrate on specifying classes and encapsulating data. It is thus possible to make the dependencies between the different parts of a program explicit and above all reduce dependencies by increasing the "localized" nature of data references (data are distributed among the objects). The functional breakdown associated with the use of the Fortran 77 or C language has often led to large sets of sub-programs using a small number of data structures. Any change to a data structure involves finding all the sub-programs that use it and rewriting them if the change affects them. As a result, developers rapidly "hard wire" the data structures, which in turn makes it very difficult to change the code. In addition, there is a tendency to restrict the number of data structures and set up all-purpose data structures (catering for what is thought to be the most general case). Adding anything that was not initially planned is very difficult. Adding a new, specialist data structure alongside an existing one involves rewriting all the associated procedures and thus duplicating code, which in turn must be validated all over again. The sub-programs are free to handle data and change them; it is hard to tell which sub-program changed a given data structure, which increases maintainability problems. Object-oriented design means that the program is built up around the data (without prejudging the manner in which they are represented) and the relationships between them. Objects communicate with one another by exchanging *messages* defined by their *interface*. Changing an object's data structure does not affect the rest of the code in any way (it is consequently not mandatory to determine the best implementation right from the start). Enhancing the interface does not have a negative impact on the rest of the code. Services can thus be added on demand, making the system extremely flexible and upgradable. An existing object can be given a speciality by *inheritance*; at the very least, the derived class offers the same services as the ancestor class (it inherits the interface from the ancestor class) so other objects will use the new class in the same way as the ancestor class. Not only does the addition of a new class not jeopardize existing code, but also it is directly compatible with it. The new class may change the way in which certain services are implemented and provide new ones. Only the methods of an object have access to the object's data, which facilitates maintainability (*encapsulation*).

1.3. The choice of C++ language

We choose to use C++ for the following reasons:

- it is an object-oriented language that supports inheritance, polymorphism and encapsulation;
- the language uses strong typing which contributes to achieving reliability objectives;
- it is a compiled language and is therefore compatible with the level of performance required;

- it can invoke sub-programs written in C and Fortran 77;
- it is a de *facto* standard, with a large installed base. This contributes to the life expectancy of the software and enables it to benefit from the development of a large number of software engineering tools and component libraries.

We consider C++ to have a big drawback in that it is implementing many different concepts by pointers. That is $A * a;$ can be used to implement associations, aggregations, arrays, ... and we don't know if a will be of type A or any of it's subtypes. We choose to provide a set of classes implementing associations, aggregations, arrays, ... so that the design will be apparent in the code. For sample, we will write "$REF<A>a;$" or "$Any<A>a;$" or "$ArrOf<A>a;$" which also enables automatic retro conception.

1.4. Building a framework for thermohydraulic

There are three main types of software developments:

- Applications Programs
  In that case everything is known at the beginning of the development, efforts are made to increase internal reuse in order not to design and implement more than what you need.
- Toolkits
  A toolkit is a set of related and reusable classes designed to provide some common functionality. Toolkits emphasize code reuse. Diffpack [4] is an example of a toolkit.
- Frameworks
  A framework is a set of cooperating classes that make up a reusable design for a specific class of software. You can customize a specific framework to a particular application by creating application-specific subclasses of abstract classes from the framework. The framework dictates the architecture of your application. Reuse in this level leads to an inversion of control between the application and the software on which it's based. When you use a toolkit, you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code it calls. In that sense Frameworks and toolkits are complementary. Not only you build applications faster as a result, but also the applications have similar structures so that they seems more consistent to their users.

We choose to design a framework for thermohydraulic.

1.5. Modular architecture

Figure 1 provides an overall view of all the main Trio_U modules. In this figure, the arrows indicate a link between two modules:

The modules on the lower part are specialized versions of the Kernel for specific physical problems. The modules in the upper part of the diagram are specialized versions of the Kernel for specific numerical methods. The modules on the left are not related to the framework and those on the right are useful for many physical problems and numerical methods. To build a concrete application (code to solve a given problem using a given numerical method) involves specializing the Kernel in two ways at the same time, with respect to the physical problem and the numerical methods. Developers who choose to introduce a new class by specializing one of the existing classes are guided by the architecture, which provides a model to follow.

2. AN INTRODUCTION TO THE FRAMEWORK

In the following, we provide an overview of the classes and their relationships without discussing C++ related issues regarding design and implementation. The layout reflects the manner in which the classes are organized into modules. Object-oriented design uses an abstract representation of the system being modeled. If the system is fairly complex, it is necessary to introduce classes representing concepts that differ a great deal and it is convenient to draw a distinction between several domains within the model. By module, we refer to a set of classes that correspond to the same domain of the modeled system, it can also be viewed as a toolkit fitting to
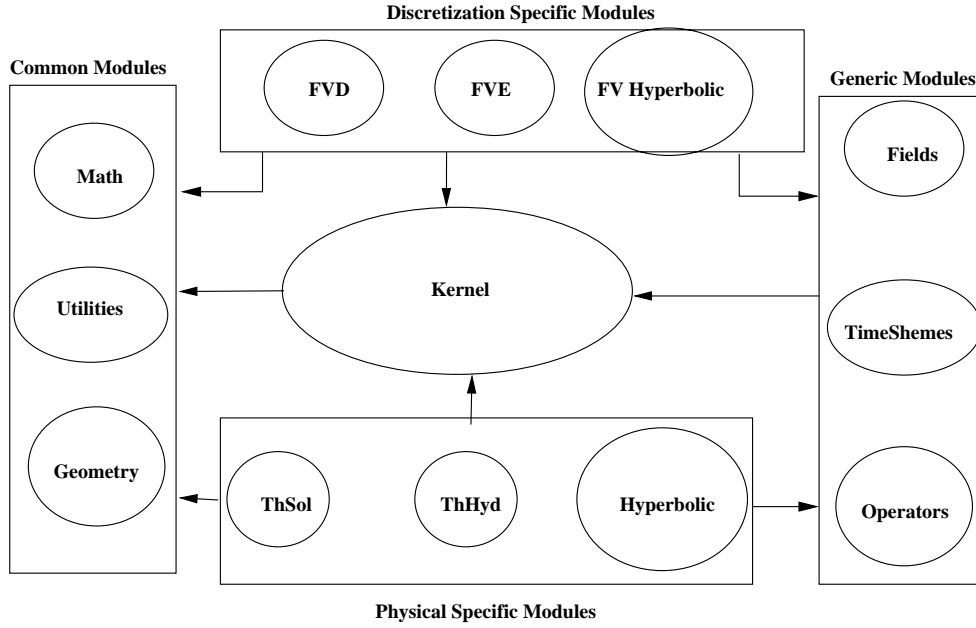
FIGURE 1. Modules.

our framework. All the classes that serve as basic bricks for building thermohydraulic applications belong to the Kernel module. The term application refers to a software sub-assembly, comprising a certain number of modules, which performs calculations in a particular field of fluid mechanics. Building an application will involve using classes belonging to the *Geometry, Fields, Time_schemes, Operators, Utilities* and *Math* modules, for example. *ThHyd, ThSol* and *Hyperbolic* modules are specialized versions of Kernel used for particular applications: *ThHyd* for the thermohydraulic of flows of incompressible fluids, *ThSol* for conduction in solids, *Hyperbolic* for flows of compressible ideal gases.

2.1. **Overview of kernel classes**

The entities in thermohydraulics are fluids and the laws and differential equations governing flow. A problem is a set of equations that may or may not be coupled. Trio_U simulates flows and consequently applies to a finite geometrical configuration. Simulation involves the notion of geometrical domains and, subsequently, the notion of boundary conditions. Trio_U is based on approximate solving of the equations and this leads to notions such as meshed domains, discrete fields, spatial discretization methods, time discretization schemes and initial conditions. The equations have one unknown (which is a scalar, a vector or a set of scalars) and consist of operators, which act on the unknown, and source terms. We have, of course, associated classes with all these notions. Figure 2 provides an overview of the classes that we created to model the above concepts as well as relationships between the classes. The notation used are UML-compliant (see [5]). More precisely an equation is supposed to be of the form:

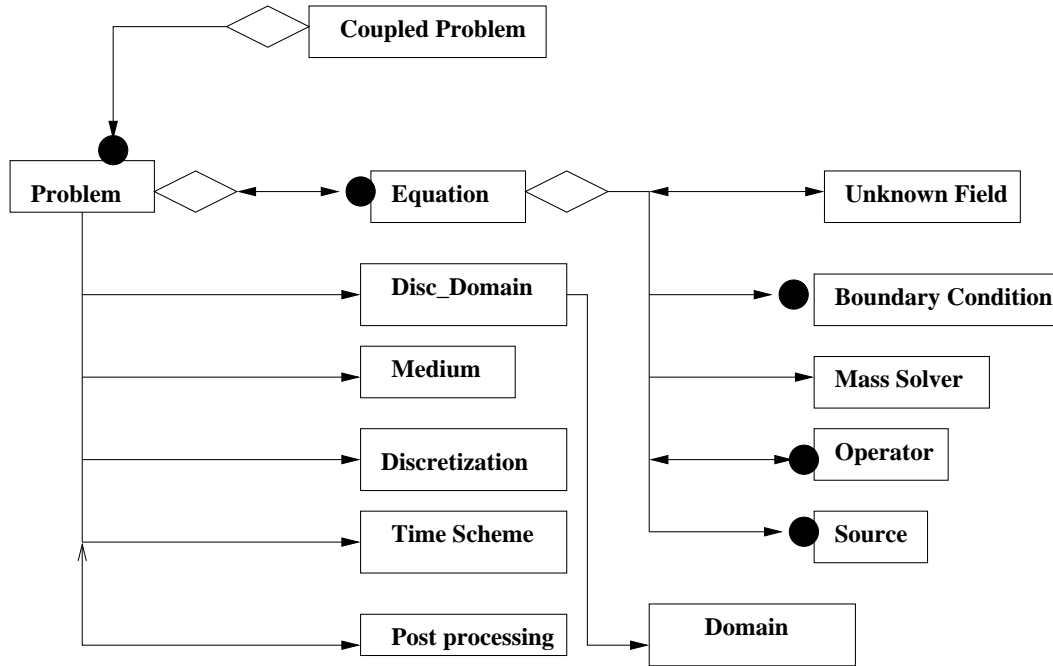$$\frac{\partial \phi}{\partial t} + \sum_i Op_i(\phi) = \sum_j f_j \tag{1}$$

FIGURE 2. Kernel.

After applying a spatial *Discretization*, we obtain:

$$M\frac{\partial \phi^h}{\partial t} + \sum_i Op_i^h(\phi^h) = \sum_j f_j^h \tag{2}$$

where M is the mass matrix -so that we will need a *Mass Solver*, ϕ^h is the *Unknown Field*, Op_i^h are discrete differential *Operators* and f_j^h are *Source* terms. A *Medium* stores the fluid (or solid) properties. These equations are associated with *Boundary conditions*. A *Problem* is a collection of such *Equations* discretized using the same *Discretization* and the same *Time Scheme* solved on the same discretized *Domain*. A *Coupled Problem* is a collection of problem differing either by their *Equations*, their *Domain*, their *Medium*, their spatial *Discretization* or their *Time Scheme*. For each *Problem* we supply some *Post Processing* tools.

2.2. **Specializing the Kernel**

Before describing in more details the main classes of the Kernel, we will show how we can specialize it for a given application. We will follow the sample of sigle-phase incompressible flows. In that case the problem writes:

$$\frac{\partial u}{\partial t} - \nabla \nu \nabla u + u\nabla u + \nabla p = g \tag{3}$$

$$\nabla . u = 0 \tag{4}$$

where ν is the viscosity and g the gravity field.

Pressure is considered as a byproduct of the divergence free velocity condition so that we can consider this equation as a special case of (2). Figure 3 shows how we can specialize the framework for that (simple) case.
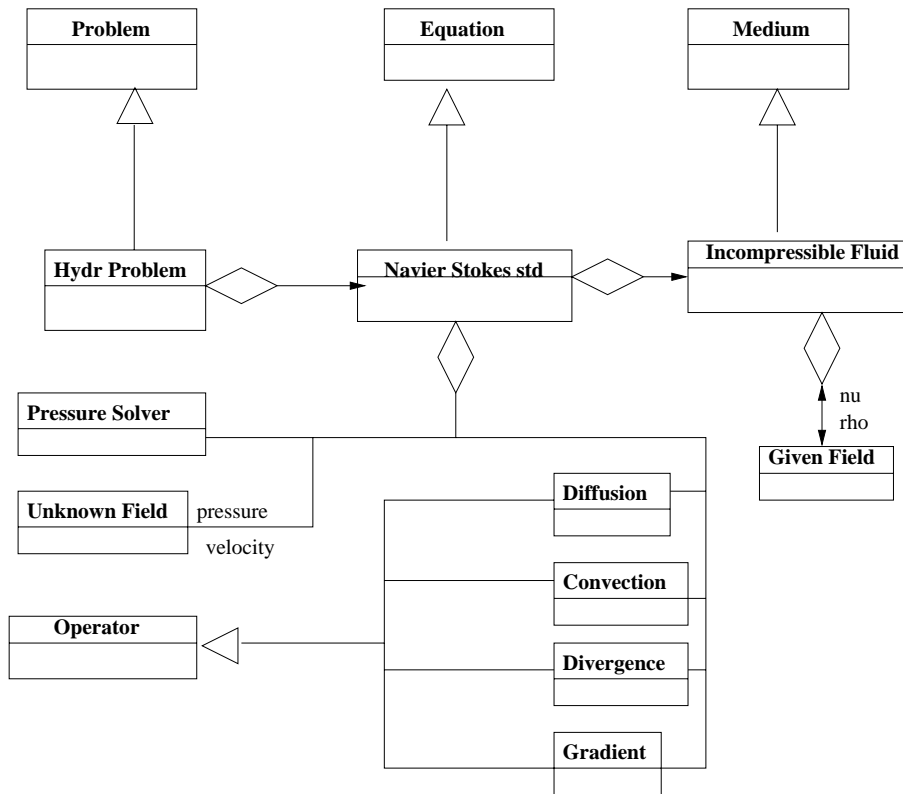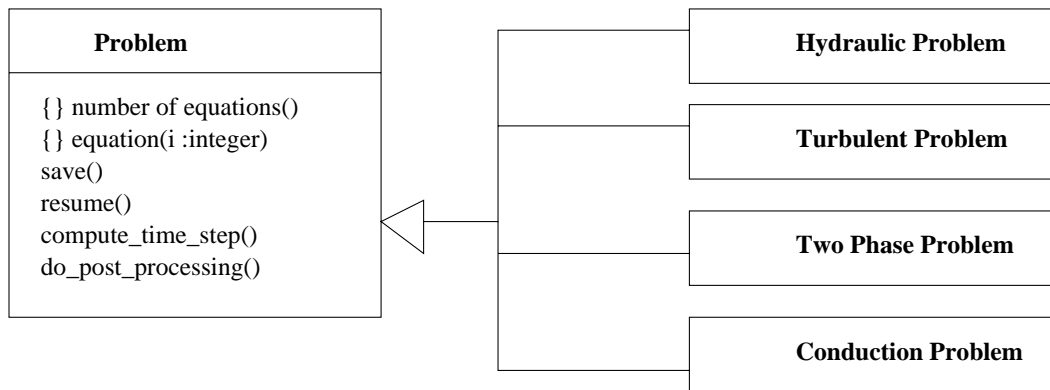
FIGURE 3. Incompressible flows.

FIGURE 4. Problem hierarchy.

2.3. Problem hierarchy

Problem is an abstract class that acts as the basic class for the hierarchy of "problem" classes. Figure 4 shows the analysis object model for the Problem class.

The Problem class has several components:

- a Disc_domain type object, which itself has a link with a Domain type object. The Domain type object represents the meshed geometry; the Disc_domain type object provides the information on the geometry and connectivity required to solve the problem with a given numerical method;
- a Postprocessing type object, which is used for implementing post-processing.

The Problem class has several associations. A Time_scheme type object, which represents the time scheme selected for the problem and a Discretization type object, which represents the spatial discretization method selected for the problem are associated with it. The Problem class is used to solve, in the most general possible way, a set of equations for a given domain. The equations that make up the problem are all discretized using the same numerical method and time scheme. The Problem class has "virtual equations", or in other words the classes derived from Problem will have one or more components that are equations. The Problem class has abstract methods that are implemented by the derived classes:

- the number_of_equations() method which returns the number of equations making up the problem;
- the equation(int i) method which returns the inth equation.

These virtual methods enable us to associate general algorithms with the Problem class. The algorithms are encapsulated in methods belonging to the Problem class; the methods may be redefined by the derived classes. Thanks to the number_of_equations() and equation() methods, Problem class operations are delegated to each equation making up the problem.

```
Problem :: calculate_time_step()
{
   double dt = time_scheme->maximum_time_step() ;
   for (int i=0 ; i<number_of_equations() ; i++)
      dt = min(dt,equation(i).calculate_time_step()) ;
}
```

2.4. Equation hierarchy

Equation is an abstract class that is used as a basis for the hierarchy of the "equation" classes.

The Equation class has "virtual operators", in other words the classes derived from Equation will have one or more components that are operators. Figure 5 shows the Equation class analysis object model. The Equation class has several components:

- a Disc_Domain object containing the boundary conditions and connectivity required to discretize the boundary conditions;
- a Mass_solver object corresponding the mass matrix associated with the selected numerical method;
- a set of Source objects to represent any source terms in the equation (this set may be empty).

The Equation class has several associations: an object belonging to one of the types derived from problem which is the problem of which the equation is a component, a Time_scheme type object which represents the time scheme selected for the problem and a Disc_zone type object which represents a discretized zone. The Equation class has abstract methods that are implemented by the derived classes:

- the number_of_operators() method which returns the number of operators that make up the equation;
- the operator(int i) method which returns the inth operator;
- the unknown() method which returns the equation's principal unknown.

A parallel may be drawn between the Problem and Equation classes; the algorithms are built using exactly the same model (with the operators in the equation playing the part of the problem's equations). These virtual methods enable us to associate general algorithms with the Equation class. The unkn_time_derivative method is the main method of the Equation class and an example of this technique:

```
Equation :: unkn_time_derivative(DoubleTab& derivative)
{
```
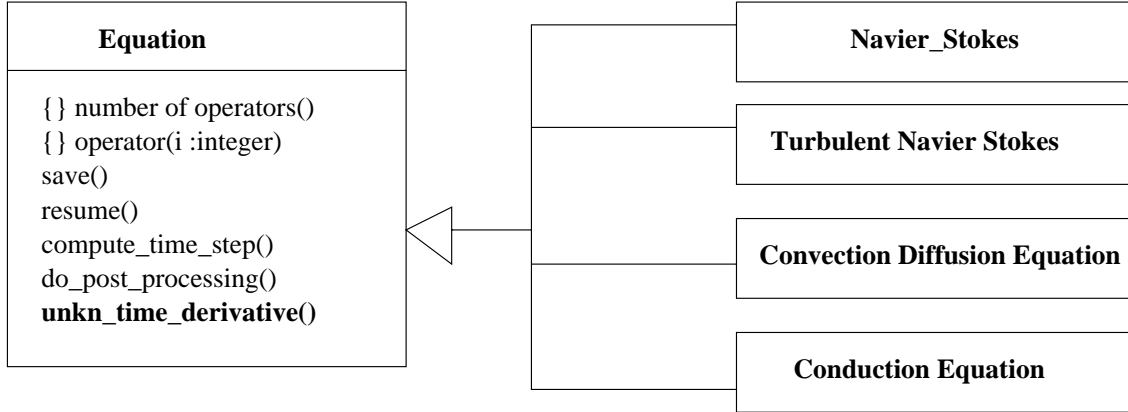
FIGURE 5. Equation Hierarchy

```
   derivative = 0 ;
   for (int i=0 ; i<number_of_operators() ; i++)
      operator(i).add(derivative) ;
   sources.add(derivative) ;
   return mass_solver.apply(derivative) ;
}
```

We have already explained that for Navier-Stokes equations, the calculation of the time derivative of the unknown (the velocity) is a special case.

After applying a spatial *Discretization*, we obtain:

$$M\frac{\partial U^h}{\partial t} + ConvDiff(U^h) + B^t P^h = F^h \tag{5}$$

$$BU^h = 0 \tag{6}$$

which leads to:

$$\frac{\partial U^h}{\partial t} = \mathcal{P}(N(U^h)) \tag{7}$$

where:

$$N(U^h) = M^{-1}(F - ConvDiff(U^h))$$
$$\mathcal{P} = Id - M^{-1}B^t E^{-1}B$$
$$E = BM^{-1}B^t.$$

The Navier_Stokes class will have its own version of the unkn_time_derivative() function (using *polymorphism*):

```
Navier_Stokes :: unkn_time_derivative(DoubleTab& acceleration)
{
// standard acceleration :
   Equation :: unkn_time_derivative(acceleration);
```

```
// Projection of acceleration on the divergence free subspace :
   divergence.add(acceleration,secmem);
   pressure_solver.solve(secmem,pressure);
   gradient.calculate(pressure,gradP);
   mass_solver.apply(gradP);
   acceleration -= gradP;
}
```

2.5. **The time schemes hierarchy**

Time_scheme is an abstract class that is used as a basis for the hierarchy of the "time scheme" classes. The class has a set of attributes that are the characteristics common to all the time schemes, both implicit and explicit, and a set of methods that are used to manage the time advance of a transient calculation. The methods are valid for all the classes derived from Time_scheme. The class has a single abstract method, do_a_time_step(Equation&). To design a class derived from Time_scheme, one must just define the do_a_time_step(Equation&) method. For example, the following algorithm, for the do_a_time_step(Equation& eq) method, corresponds to the Explicit_Euler_Scheme class:

$$U^{n+1} = U^n + \Delta t \left(\frac{\partial U}{\partial t} \right)^n \tag{8}$$

```
Explicit_Euler_Scheme :: do_a_time_step(Equation& eqn)
{
   DoubleTab& future=eqn.unknown.future();
   eqn.unkn_time_derivative(future);
   future*=dt;
   future+=eqn.unknown.present();
}
```

where future stands for U^{n+1} and present for U^n.

2.6. **Field hierarchy**

Field is an abstract class that is used as a basis for the hierarchy of the "field" classes. Figure 6 shows the field class analysis object model.

The field class possesses the following attributes: a name, a number of components, a nature (the field is scalar, vectorial or set of scalars), a name and a unit of measurement for each of its components and a time value.

The field class carries a set of abstract operations value_at (or values_at), thanks to which it is possible to access the value of the field at any point (or set of points) on the calculation domain. It also includes the abstract method inject(field) which injects the field used as a parameter into the current instance. For the fields, two other hierarchies may be distinguished within the main hierarchy:

- the hierarchy of the fields which represent an unknown. The Unkn_Field class is used to represent fields that are calculated by solving an equation, so it has an association with the equation used to calculate the field. This class has a Wheel component that represents the values at different times. The number of elements in the Wheel depends on the time scheme being used to discretize the equation (Explicit_Euler needs the values of the field, at two different times). It also has a virtual method "nodes_coordinates()" returning the coordinates of the discretization's nodes;
- the hierarchy of the fields which represent a given field. The Given_Field class is used to represent the fields that are functions of the spatial coordinates, the time and eventually one or more unknown fields.
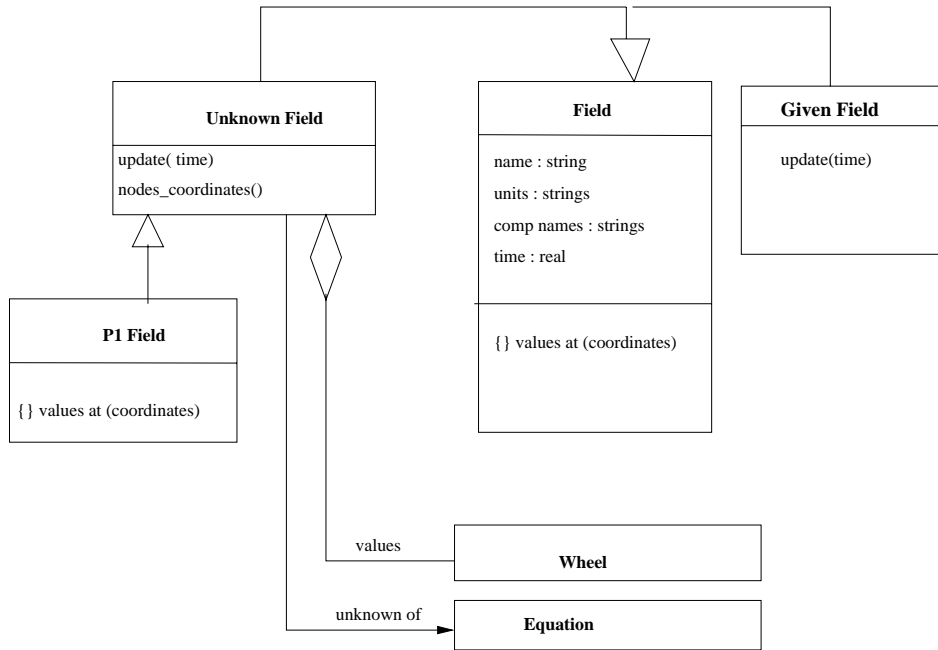
FIGURE 6. Field

The inject(field), value_at() and values_at() methods play a very important role in the writing of generic algorithms using the fields. For example the Lagrange interpolation can be coded very simply:

```
Unkn_Field::inject(const Unkn_Field& f)
{
values=f.values_at(nodes_coordinates());
}
```

The only methods that need to be redefined by derived classes are the ones involving localization of the degrees of freedom of the field (values_at, and nodes_coordinates()).

It should be noted that another (and maybe better) conception would have been to distinguish mainly between discrete fields and continuous ones.

2.7. Discretization and disc_domain hierarchy

Discretization is an abstract class that is used as a basis for the hierarchy of classes that represent a spatial discretization methods. The discretization class and its derivatives are classes comprising methods only and the methods are used to discretize the various objects involved in solving problems. The Discretization is responsible of the creation of the connectivities and geometrical information (stored in a Disc_Zone object) needed by the method and deduced from the Domain. It is also responsible of the typing of the Unknown_Fields. It can be considered as an Object Factory pattern (see [3]).

The Disc_Domain classes have fundamentally one abstract method: "discretize()" which corresponds to the creation of all the geometrical and connectivity information related to the numerical method.

2.8. Operator hierarchy

Operator is an abstract class that is used as a basis for the hierarchy of the "operator" classes. The operator class is used to implement the $Op(U_h)$ term. We have introduced classes derived from the operator class to represent the various sorts of term involved in the equations.

- Conv_operator is the basic class for operators that represent a convective term.
- Diff_operator is the basic class for operators that represent a diffusive term.
- Grad_operator is the basic class for operators calculating the gradient of an unknown scalar field.
- Div_operator is the basic class for operators calculating the divergence of an unknown vectorial field.

The concrete classes are specialized versions of these basic classes, designed for a given numerical method and, if appropriate, a given calculation option. We shall use the convective term as an example. The FVD_Upstream_Conv_op class which is derived from the FVD_Conv_op class corresponds to using the upstream scheme for the convective operator ($\vec{u}\vec{\nabla}\phi$) for Finite Volumes Difference.

3. Parallelism

In this section, we give a brief description on how Object Oriented Design can simplify the task of building a parallel application. For more details see [1] and [2].

3.1. **Parallel computing.**

Given that we must be able to finely simulate flows in complex, 3D geometries, discretization of the studied domains generally leads to grids comprising several million elements. To solve problems of this sort multiple-processor machines, and in particular massively parallel machines with distributed memory, are needed.

The most commonly used approach when parallelizing scientific code is to start with sequential code and add directives, or primitives, at appropriate points making it possible to parallelize the code. The major drawback with this approach is that for each new development, the work of parallelization has to be carried out again for the newly added module. This often results in two versions of the same code – one sequential, the other parallel – with the obligation to call in a "parallel processing specialist" each time the parallel version of the code needs to be updated. To avoid this type of problem, we opted to design it as a parallel application. From the start of code design, all the building bricks were designed for parallel processing. The ultimate objective is to make the parallelism completely transparent, not only for end users but also for the majority of co-developers. In other words, users must find the same behavior, operating mode and application usage, regardless of the target machine (single or multiple processor). In addition, a co-developer wishing for instance to add a new physical model must not need to bother about parallelizing it. We set the following objectives:

- transparent parallel processing;
- only one set of code may be used at any one time on all architectures;
- transparent for end-user: parallel calculation is triggered by an option in the data set;
- transparent for the co-developer: any new development will be implicitly parallel;
- physical quality of results: the same physical results should be obtained with parallel and sequential processing;
- efficiency and performance: this objective is almost self-evident in the case of parallel processing. However, it must be balanced with the preceding objectives and a compromise must be established between performance and portability, reusability, modularity, quality and transparency;
- portability: the same code must be able to run on any architecture, from a PC to a massively parallel computer with distributed memory. The following choices were made with respect to parallelism models;
- parallelism model: data parallelism thanks to an initial breakdown of the calculation domain. This is a natural choice for this type of application;
- programming model: SPMD (Single Program Multiple Data) — the same program is executed by all the processors. This choice is a direct result of the previous one;
- explicit message exchange (use of libraries such as PVM or MPI). This choice is not only dictated by the need for high performance, but also the desire to take parallel processing to the roots of the code.

The previous choices are fairly general and widely used. They do not impact directly on the design of the code. The main objectives that guided the fundamental design decisions were the transparent nature of the parallel processing and the quality of the results obtained. We have not imposed the use of domain decomposition
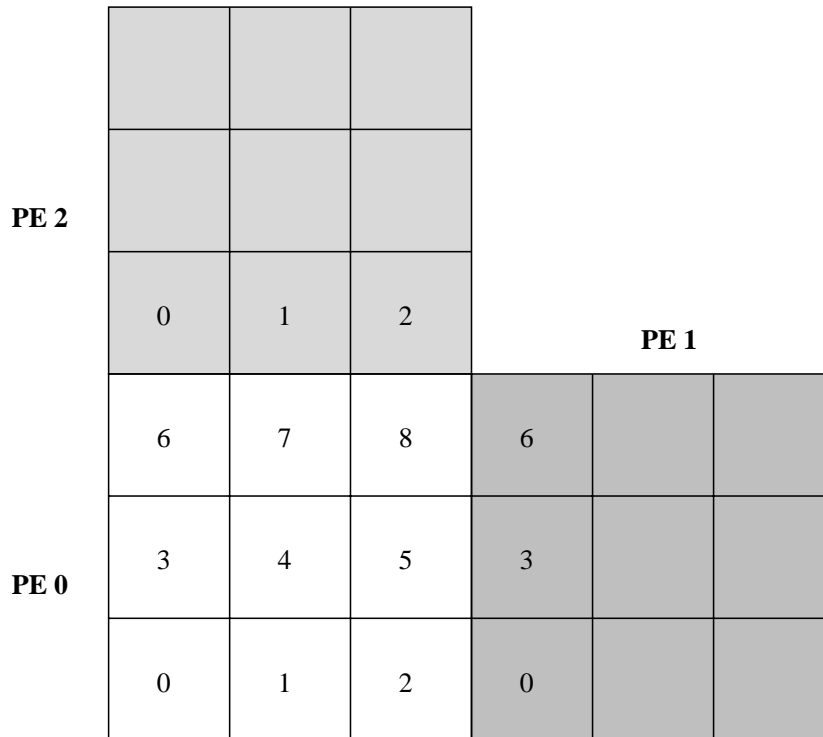
FIGURE 7. Domain splitting: PE0 stores $((v_0^0, v_1^0, ..., v_8^0), (v_0^1, v_3^1, v_6^1), (v_0^2, v_1^2, v_2^2))$.

methods because the numerical results obtained in sequential or parallel computations won't be identical. Only one physical problem is addressed, as with sequential processing, and it is solved cooperatively by the various processors. To achieve this, each of the processors must have a copy of values computed by other processors. We apply the "owner computing rule", in other words a processor does not calculate values localized on meshes or faces that do not belong to its sub-domain. In the following example (see Fig. 7), we have three sub-domains.

Suppose we want to compute the gradient at the mesh edges of a constant by element field using the simplest finite difference scheme $\frac{p_{i+1} - p_i}{\Delta x}$. We see that processor 0 will need 3 values from processor 1 (at element 0, 3 and 6) and 3 from processor 2 (at element 0, 1 and 2).

We have consequently designed objects that make it possible to store not only the values on a given calculation domain, but also a copy of the values from neighboring domains. The array has a real part containing the values on the sub-domain under consideration, and a virtual part, which contains a copy of the values on the neighboring sub-domains. It also has a description of the values to be send to and of the values he will receive from other processors. All these information are provided when an Unkn_Field is constructed. For example a P_0 field will create a distributed vector that knows how to exchange values at elements for a given stencil. Moreover the Unkn_Field class provides an update() method which is called (at least) at the end of each time step and which will also update the virtual values. The description of the virtual part of the value vector is also used for the resolution of the linear systems. The update method is a generic one:

```
Unkn_Field::update()
{
   wheel.roll();
   values.exchange_data();
}
```
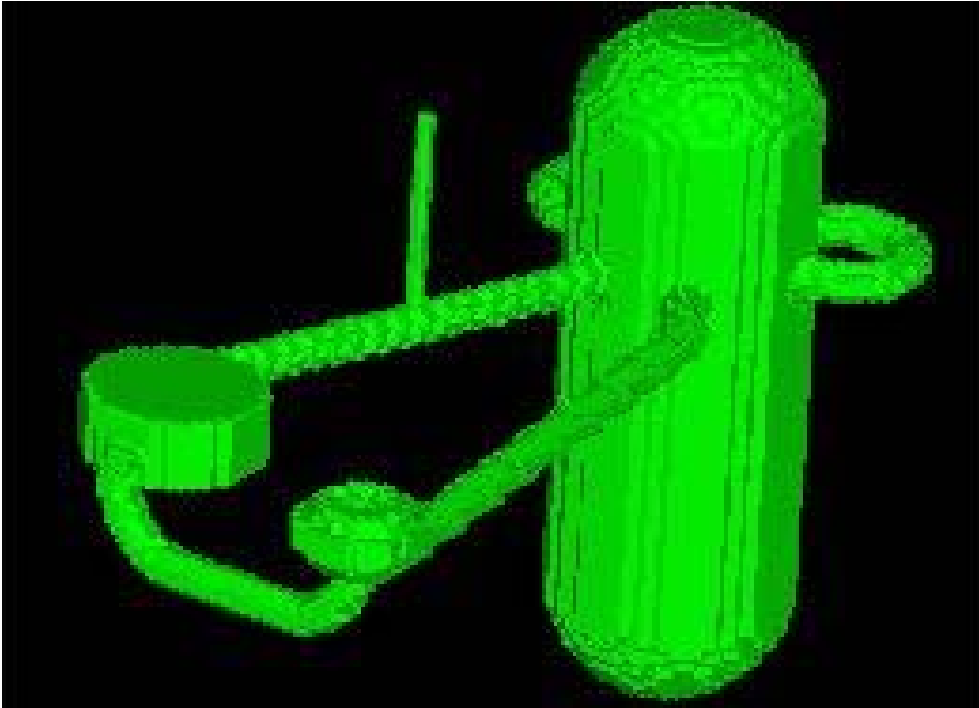
FIGURE 8. Wessel.

3.2. Parallel I/o's

We have required that every object in the application is able to read an write itself on a stream. A communication chanel being just a type of stream, every object can be sent from one processor to another. Object Oriented designs is also very convenient for dealing with parallel I/o's. We have developed a library of IO classes enabling us to diffuse users's data from a single file to each processors and to collect numerical results from each processor to a single file for exemple.

4. Calculation, performance and software environment

4.1. Performance

It is always tricky to discuss the raw performance of code. Simulation time depends to a very large extend on the case under study and the physical models, numerical schemes and hardware being used. However, a recurrent question concerns the performance of code using an object-oriented language compared with a functional programming language such as Fortran. We have made a close comparison of the performance of Trio_U and its predecessor Trio_VF, which was programmed in Fortran. This study has enabled us to draw the following conclusions:

- Object-based mechanisms do not penalize performance if they are used at a certain level in the code, on the contrary. However, they must not be used in the lowest calculation loops.
- Time is spent solving linear systems, in which optimized routines (such as BLAS routines) are invoked.
- The performance differences observed between Trio_U and Trio_VF with respect to solving systems (around 40% overcost for Trio_U) are related to the manner in which the matrices are stored. Trio_VF uses IJK data structures leading to direct addressing while Trio_U uses indirect addressing. For the rest of the code, Trio_U yields higher performance (from 30 to 40%).
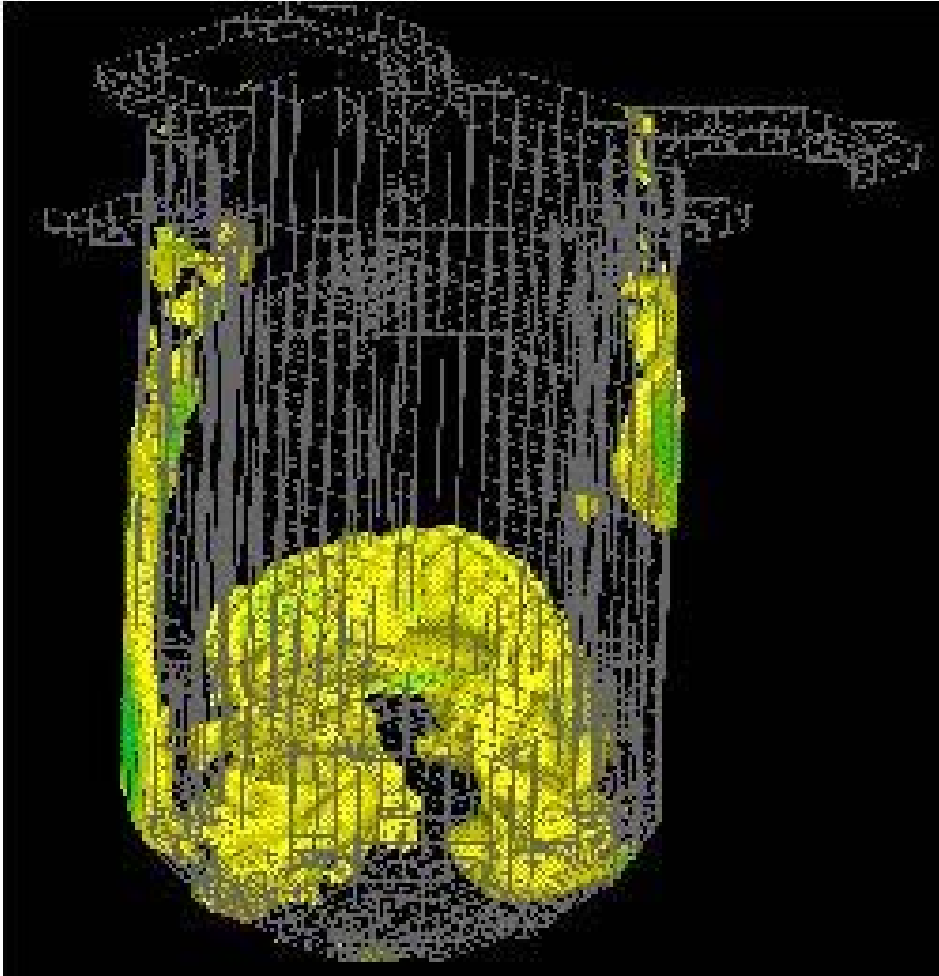
FIGURE 9. Clear water concentration in the core.

4.2. **Industrial calculations**

The following example concerns the calculation of the transient thermal hydraulics in the primary cooling system of a 900 MW PWR (Pressurized Water Reactor) using Trio_U version 1 (see Figs. 8 and 9). The simulation focuses on the dilution in the primary system of a plug of clear water (water whose boron concentration is lower than the rated concentration in the primary system). In short, this is a thermohydraulic problem involving the transport of a concentration. The following choices were made regarding the representation of the primary system:

- the pressure vessel was modeled approximately, with the downcomer, inlet plenum, core and outlet plenum;
- one of the three cooling system loops was modeled approximately with the hot leg, to which a pressurizer, the bottom of the steam generator, the crossover leg, pump and cold leg were connected;
- the other two loops were represented in less detail by the equivalent shunts (the flow rates distributed over the three loops were modeled).

The primary cooling system was closed, initially at rest, in an isothermal state, and the pressure was determined by the pressurizer. The crossover leg was filled with clear water. Simulation started at the moment when the

pump was started. Calculation characteristics were as follows:

- 400 000 cubic elements;
- incompressible thermohydraulics;
- simulated time: 22 s; after this lapse of time, almost all of the clear water plug had crossed the core.

The results of the simulation show the position of the clear water all over the primary system.

5. CONCLUSION

We have shown in this article how object-oriented design has enabled us to develop an open, modular software architecture. The current version of Trio_U, which was written entirely in C++, is used for industrial calculations in single-phase thermohydraulics and research in both single-phase turbulent thermohydraulics and two-phase direct simulation. It has demonstrated its effectiveness (open, modular architecture) for software yielding performance equivalent to software written with Fortran.

REFERENCES

[1] C. Calvin and P. Emonot, The parallelism in Trio-Unitaire. *Actes de la conférence NURETH 8*, Kyoto Japan, October (1997).
[2] C. Calvin and P. Emonot, The Trio-Unitaire Project: A parallel CFD 3-Dimensional Code. *Actes de la conférence* IS-COPE (1997).
[3] M. Farvacque, O. Cueto, P. Emonot and F. Barre, A new generation of ThermalHydraulics computer code. *Actes de la conférence NURETH 8*, Kyoto Japan, October (1997).
[4] J. Rumbaugh *et al.*, *Object Oriented Modeling and Design.* 2nd ed., Prentice Hall (1991).
[5] B. Stroustrup, *The C++ programming language.* 2nd ed., Addison Wesley (1992).