Scientific Programming 14 (2006) 111–139
IOS Press

Reusable object-oriented solutions for numerical simulation of PDEs in a high performance environment

Andrea Lani[a,*], Tiago Quintino[a,b], Dries Kimpe[b,c], Herman Deconinck[a], Stefan Vandewalle[b] and Stefaan Poedts[c]
[a]*Von Karman Institute, Aerospace Dept., Chaussee de Waterloo 72, B-1640 Sint-Genesius-Rode, Belgium*
[b]*Catholic University Leuven, Computer Science Department, Celestijnenlaan 200A, B-3001 Leuven, Belgium*
[c]*Catholic University Leuven, Center for Plasma-Astrophysics, Celestijnenlaan 200B, B-3001 Leuven, Belgium*

Abstract Object-oriented platforms developed for the numerical solution of PDEs must combine flexibility and reusability, in order to ease the integration of new functionalities and algorithms. While designing similar frameworks, a built-in support for high performance should be provided and enforced transparently, especially in parallel simulations. The paper presents solutions developed to effectively tackle these and other more specific problems (data handling and storage, implementation of physical models and numerical methods) that have arisen in the development of COOLFluiD, an environment for PDE solvers. Particular attention is devoted to describe a data storage facility, highly suitable for both serial and parallel computing, and to discuss the application of two design patterns, Perspective and Method-Command-Strategy, that support extensibility and run-time flexibility in the implementation of physical models and generic numerical algorithms respectively.

1. Introduction

COOLFluiD (Computational Object-Oriented Library for Fluid Dynamics) [13] is a framework originally developed to solve fluid-dynamics related PDEs (Partial Differential Equations) on unstructured grids by means of state-of-art numerical techniques.

Similarly to many other scientific platforms, such as *DiffPack* [4], *Overture* [10], *ELEMD* [16], *MOUSE* [24] and *OpenFoam* [25], COOLFluiD is implemented in C++, in order to take advantage of both classical object-oriented (OO) techniques [15,20,21] and generic template-based programming [1,22,23]. Effort has been devoted in searching and implementing long term architectural solutions, in order to allow both run-time flexibility and high performance, as required by the target applications.

In particular, while designing the kernel interfaces and components of COOLFluiD, no strict a priori assumption has been made on the kind of PDE solving algorithms and discretization techniques to support. Consequently, a multitude of numerical solvers for unstructured grids can be implemented within the platform. This algorithm-independent design policy marks in fact a sound difference between COOLFluiD and the majority of other available OO frameworks for Computational Fluid Dynamics (CFD), which are usually tailored towards specific space discretization schemes (e.g. Finite Volume for [24,25], Finite Volume and Finite Difference on block structured meshes for [10], cell-vertex schemes for [16], etc.).

*Corresponding author. Tel.: +3223599611; Fax: +3223599600; E-mail: lani@vki.ac.be.

ISSN 1058-9244/06/$17.00

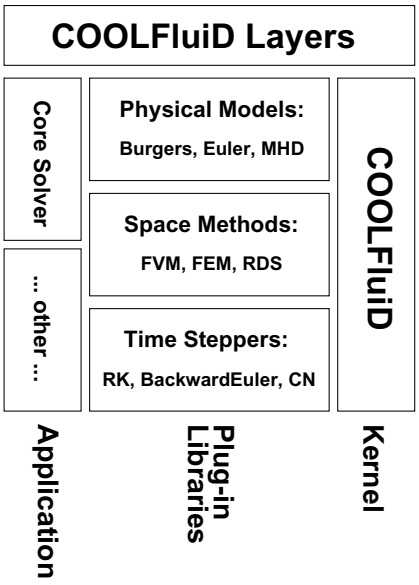

Fig. 1. Multi-layer architecture of COOLFluiD.

1.1. COOLFluiD architecture

COOLFluiD consists of a collection of dynamically linked libraries and application codes, organized in a multiple layer structure, as shown in Fig. 1. Two kinds of libraries are identifiable: kernel libraries and modules (*plug-ins*). The former supply basic functionalities, data structure and abstract interfaces, without actually implementing any scientific algorithm, while the latter add effective simulation capabilities. The framework was developed according to a *plug-in policy*, based on the self-registration [3,13] and self-configuration techniques [13], which allow new components or even third party extensions to be integrated at run-time, while only the API has been exposed to the developers. From top to bottom of the COOLFluiD architecture, one finds, progressively, more functional layers: the kernel defines the interfaces, the modules implement them and the application codes can select and load on demand the libraries needed for an actual simulation. This multi-layer structure guarantees flexibility and modularity at the highest design level and helps managing the development process, since the layers automatically reflect different implementation levels.

1.2. Outline

Designing a framework for generic PDE solvers is not a trivial task by itself and additional complexity comes in our case from planning to deal efficiently with arbitrary numerical algorithms and discretization schemes with different data structures.

According to a simplified but sufficiently complete conceptual view, we can consider a numerical solver for PDEs to be made by a limited number of independent building blocks: some mesh-related data, typically numerics-dependent, that represent a discretized view in terms of both geometry and solution of the computational domain; an equation set describing the physical phenomena to study; a group of interacting numerical algorithms to solve the PDEs on the given discretized domain.

We can therefore identify three issues of fundamental importance for developing an environment for solving PDEs:

– handling and storage of bulk serial and distributed data;
– implementation of a physical model;
– implementation of a numerical method.

In our opinion, a good OO design should be able to tackle these three aspects orthogonally, independently from one another as much as possible, in order to minimize coupling and maximize the reusability of the single components. To this aim, a clear separation between physics and numerics should be enforced, the first being the description of the properties, constitutive relations and quantities that characterize the equation system, the second providing the mathematical tools (algorithms, algebraic and differential operators) to discretize and solve those equations.

This article will focus on analyzing each one of the three above-mentioned problems in a separate Sections (2, 3, 4), and on presenting corresponding possibly effective solutions, as they were implemented in COOLFluiD. Some illustrative Examples (3.1, 4.1) will also be given in order to show the concrete applicability of the proposed ideas.

2. Data handling and storage

PDE-driven simulations typically involve a considerable amount of data related to the discretization of the computational domain, such as coordinates, state variables, geometric entities, etc., all grouped inside container objects (arrays, lists . . .) on which numerical actions are performed. Additional complexity appears when computing on unstructured grids, as in the case of COOLFluiD, since different numerical methods can require the storage and usage of different kinds of connectivity information. As a result, genericity in both quantity and types of data must be addressed.

Moreover, in a scalable and safe design of the data handling, framework components should be able to share data without violating encapsulation.

In a high performance environment, developers of new modules should be allowed to elect newly defined data types to be stored, handled efficiently, and shared in parallel communication. Ideally, local and distributed data should be treated uniformly from the developer's point of view, and support for different communication strategies (message passing and shared memory) should be offered.

2.1. Data storage

In COOLFluiD, all data whose size scales with the problem complexity are encapsulated by a `MeshData` object, that, in particular, aggregates and offers safe access to a number of underlying instances of `DataStorage`.

```
template <class TAG> class DataStorage {};
```

DataStorage is defined as a template empty class parameterized with a lightweight tag policy, `TAG`, which is meant to specify the actual parallel implementation or communication type (e.g. `LOCAL`, `MPI`, `PVM`, `SHM`):

```
class LOCAL {}; // serial communication
class MPI   {}; // MPI-based parallel communication
class PVM   {}; // PVM-based parallel communication
class SHM   {}; // shared memory communication
```

As explained in [12], these tags do not appear explicitly in the code, but they are aliased at compile time to more general concepts, such as `GLOBAL` for intra-communication or `REMOTE` for inter-communication, according to the available models and the user's preferences:

```
#ifdef HAVE_MPI
typedef MPI GLOBAL;
#elif HAVE_PVM
typedef PVM GLOBAL;
#else
typedef LOCAL GLOBAL;
#endif

#ifdef HAVE_SHM
```

```
typedef SHM REMOTE;
#else
typedef LOCAL REMOTE;
#endif
```

In COOLFluiD, a parallel layer is defined with the task of concealing implementation details that rely upon specific parallel paradigms. To this aim, only the aliased tags are used outside the parallel layer, i.e. in the numerical modules of the framework.

To make DataStorage instantiable, explicit template specializations [20,22] must be defined, one for each native tag class.

2.1.1. Local data storage

The class definition of `DataStorageLocal` is reported:

```
template<>
class DataStorage<LOCAL> : public DataStorageInternal<Evector>{};
```

DataStorage defines the underlying storage type and inherits the interface of the parent class `DataStorageInternal`:

```
template <template <class T> class STYPE>
class DataStorageInternal {
public:
  // Creates and initializes a storage
  template <class T> DataHandleInternal<STYPE, T> createData
  (string name, CFuint size, T init = T())
    throw (StorageExistsException);

  // Deletes a storage and frees its memory
  // Returns the number of deleted storages
  template <class T> CFuint deleteData(string name)
    throw (NoSuchStorageException);

  // Gets an existing storage
  template <class T> DataHandleInternal<STYPE, T> getData
  (string name) throw (NoSuchStorageException);

private:
  /// map that stores the pointers to arrays of data
  std::map<string, void*> m_dataStorage;
};
```

DataStorageInternal is parameterized with the storage type, `STYPE`, and the derived DataStorageLocal specifies the latter to be an `Evector` [11], which offers an improved and more efficient implementation of `std::vector` [6, 20,21]. In particular, `Evector` supports back insertion/deletion of elements in constant time and on-the-fly memory allocation to accommodate new entries with minimal overhead.

DataStorageInternal behaves as an archive where arrays of data with a certain type and size are registered under a user-chosen name in an associative container like a `std::map` [20,21]. Pointers to the arrays in question are statically casted to `void`$*$, in order to let coexist different types in the same instance of DataStorageInternal. No built-in garbage collection or deletion policies based on some sort of reference counting are automatically provided by DataStorageInternal. This fact implies that data created with `createData`() must be consistently deleted with an explicit call to `deleteData`(): ad-hoc *setup* and *unsetup* `Commands` fulfill this task in each numerical module (see 4.1).

Exceptions are thrown if client code attempts to register twice data with the same name or to delete unavailable data. In any case, just before the end of the simulation, a general clean-up is performed by MeshData and all the still reachable entries in all the instantiated DataStorages are removed.

2.1.2. MPI data storage

DataStorage, which is a proxy for DataStorageInternal, provides a uniform, user-friendly interface to create and manage arrays of generic data types, that are meant to be shared among different numerical components. Moreover, DataStorage offers the same interface to treat both local and distributed data uniformly, even though the actual implementation of its member functions depends on the tag class. In case the `MPI` tag is used, `DataStorageMPI` is defined as follows:

```
template<> class DataStorage<MPI> {
public:
  // constructor accepting a MPI communicator
  DataStorage (MPI_Comm communicator);

  // Creates and initializes a storage
  template <class T> DataHandle<MPI,T> createData
  (string name, CFuint size, T init = T())
    throw (StorageExistsException);

  // Deletes a storage and frees its memory
  // Returns the number of deleted storages
  template <class T> CFuint deleteData(string name)
    throw (NoSuchStorageException);

  // Gets an existing storage
  template <class T> DataHandle<MPI,T> getData
  (string name) throw (NoSuchStorageException);

private:
  // map that stores the pointers to arrays of data
  std::map<string, void*> m_dataStorage;
  MPI_Comm m_communicator; // communicator
};
```

DataStorageMPI does not derive from DataStorageInternal, but provides its own implementation of the same interface, so that the actual differences between one DataStorage or another remain completely invisible to the client code. The main difference between DataStorageMPI and DataStorageInternal is given by the presence of a *communicator*, needed by `ParVectorMPI`, which is the container type for the data to be registered in DataStorageMPI.

2.2. Data handle

Once registered in either a local or MPI DataStorage, data can be accessed through a smart pointer, `DataHandle`, that ensures safety by preventing data array from being accidentally deleted. It also provides inlinable accessor/mutator functions for the individual array entries and offers some additional functionalities. We present the class definition of DataHandle and its partial template specializations when the tag class is set to `LOCAL` or `MPI`:

```
template <class TAG, class T> class DataHandle {};

template <class T> class DataHandle<LOCAL,T> :
         public DataHandleInternal<Evector,T> {
  typedef DataHandleInternal<Evector,T> BaseClass;
  typedef typename BaseClass::StorageType StorageType;
public:
```

```
  // constructors
  DataHandle(StorageType *const ptr) : BaseClass(ptr) {}
  // overloaded and copy constructors, assignment operator ...

  // begin and end the synchronization
  void beginSync() {}
  void endSync() {}

  // add non updatable ghost points
  CFuint addGhostPoint (CFuint globalID) {}

  // add local point and return the index of the new point
  CFuint addLocalPoint (CFuint globalID)
  {return BaseClass::m_ptr->increase();}

  // reserve size capacity
  void reserve(CFuint n) {BaseClass::m_ptr->reserve(n);}

  // build table of ghost points to use during synchronization
  void buildSyncMap() {}

  // get the global size of the underlying data array
  CFuint getGlobalSize() const {return BaseClass::m_ptr->size();}
};

template <class T> class DataHandle<MPI, T> :
         public DataHandleInternal <ParVectorMPI,T> {
  typedef DataHandleInternal<ParVectorMPI,T> BaseClass;
  typedef typename BaseClass::StorageType StorageType;
public:
  // constructors
  DataHandle(StorageType *const ptr) : BaseClass(ptr){}
  DataHandle(void* ptr) : BaseClass((ParVectorMPI<T>*) ptr){}
  // overloaded and copy constructors, assignment operator ...

  // begin and end the synchronization
  void beginSync() {BaseClass::m_ptr->beginSync();}
  void endSync() {BaseClass::m_ptr->endSync();}

  // add a non-updatable ghost point
  CFuint addGhostPoint (CFuint globalID);

  // add local point and return the index of the new point
  CFuint addLocalPoint (CFuint globalID);

  // reserve size capacity
  void reserve(CFuint n) {BaseClass::m_ptr->reserve(n);}

  // build table of ghost points to use during synchronization
  void buildSyncMap() {BaseClass::m_ptr->buildGhostMap();}
```

```
  // get the global size of the underlying data array
  CFuint getGlobalSize() const
  {return BaseClass::m_ptr->getGlobalSize();}
};
```

DataHandleMPI declares functions that deal with the synchronization and handling of data belonging to the overlap region. The actual synchronization and communication is delegated to the acquainted parallel vector, ParVectorMPI, in which all MPI calls are encapsulated. As shown above, the same functions must be implemented also in the DataHandleLocal in order to allow all the code to compile even without having MPI, that is when the `GLOBAL` tag is aliased to `LOCAL`. For sake of completeness, we report the definition of `DataHandleInternal`, from which both the above presented DataHandles derive:

```
template<template <class T> class STYPE, class T>
class DataHandleInternal {
public:
  typedef STYPE<T> StorageType;

  // constructor
  DataHandleInternal(StorageType *const ptr) : m_ptr(ptr){}
  // copy constructors, assignment operator ...

  // overloading of subscripting operator
  T& operator[] (CFuint idx) {return (*m_ptr)[idx];}

  // overloading of operator ()
  T& operator() (CFuint i, CFuint j, CFuint stride)
  {return (*this)[i*stride+j];}

  // ...
protected:
  StorageType* m_ptr; // pointer to the array to be handled
};
```

2.3. Current and future applications

The parallel implementation of data handling and storage that has been described so far is currently available only for standard MPI [14], even though the use of other libraries for message passing communication like PVM [8] could be easily integrated. In the latter case, a new set of template specializations (`DataHandlePVM`, `DataStoragePVM` and `ParVectorPVM`) with specific PVM bindings would be required, with no implications on the client code of DataStorage, that would keep on dealing with `LOCAL` and `GLOBAL` data.

The described scheme allows the user to combine different tagged models within the same run: a `LOCAL` (serial) and a `GLOBAL` (MPI) model coexist in a parallel multi-processors SPMD (Single Program Multiple Data) COOLFluiD simulation.

The MeshData object aggregates two instances of DataStorage, one for `LOCAL` data and one for `GLOBAL` data. The former includes not only data that are created by numerical modules and that are not meant to be shared among different processors, but also a local representation of some distributed data. An example is given by storages of `State` and `Node` objects, which represent solution state vectors and geometric coordinates, i.e. the degrees of freedom of the computational grid. States and Nodes are both proxies [7] for tiny arrays of floats, on which convenient symbolic mathematical operations can be performed efficiently using the Expression Templates technique [9,22, 23]. The raw memory of these tiny arrays, which are interfaced by States (or Nodes) objects, belongs to a single `GLOBAL` preallocated unidimensional ParVectorMPI of floats, on which synchronization and communication are performed. During the computation, the synchronization and communication processes involve only a relatively

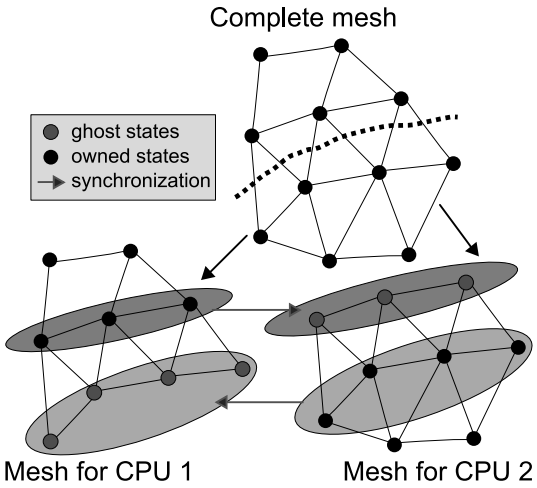

Fig. 2. Cell-wise mesh partitioning that shows updatable and ghost states in the overlap region.

small fraction of data, namely only the updatable and non updatable States and Nodes which belong to the overlap region in each process. The size of the overlap region is determined by the needs of the selected space discretization algorithm. The mesh is partitioned on a cell base by means of a polymorphic *MeshPartitioner* object: a self-rolled random partitioner and wrappers for *METIS* and *ParMETIS* libraries [18] are currently available. The overlap region is normally chosen to include all the vertex neighbors of the local cells attached to the partition boundary in each process, as it is illustrated by Fig. 2. This choice allows numerical algorithms to work solely on `LOCAL` data which already include the full computational stencil, without needing parallel communication, except at the end of each physical time step, or pseudo time step, when synchronization of remote data is needed and so is access to the `GLOBAL` data storage.

The next challenge for COOLFluiD is to deal with MPMD (Multiple Program Multiple Data) schemes in loosely coupled multi-physics applications, such as Aeroelasticity, where the equations of Fluid Dynamics are solved weakly coupled with the Elasticity equations and the computational mesh is deformed accordingly.

At the time of the writing of this paper, hybrid parallel simulations are being investigated. As a preliminary step, intra-simulation communication will be based on shared memory, which corresponds to the already mentioned `SHM` tag in the COOLFluiD terminology, and inter-simulation communication will be tackled with the already available MPI implementation. The final goal is to perform multi-physics computations in which both intra-communication among processes (processors) associated to a certain group (corresponding to a simulation on a certain domain, physics and numerics) and inter-communication among different groups will be handled by MPI (or PVM or an hybrid of the two).

In order to allow the coexistence of shared memory and MPI communication, an additional DataStorage for `REMOTE` data will be added in MeshData and additional DataStorageSHM, DataHandleSHM and ParVectorSHM will be implemented. Furthermore, the concept of *groups* needs to be introduced in the design.

2.4. Performance issues and results

The scheme and implementation that have been presented so far do not allow run-time selection of the parallel models or paradigms, but this is not really a limitation, since compile time selection offers a much better support for aggressive optimization (e.g. inlining) and high performance, helping to avoid any run-time overhead that could otherwise very likely compromise the overall efficiency. In order to make a realistic quantitative example to demonstrate this, we have tested a polymorphic DataHandleInternal with virtual accessor and mutator functions for the overloaded [] and () operators. As a result of this simple experiment, an increase of global computational time of approximately 15% has been observed in all testcases, due to the high call frequency of those tiny virtual functions, that otherwise could have been easily inlined by any available C++ compiler. Therefore, the idea of supporting run-time selection of parallel models has been discarded.

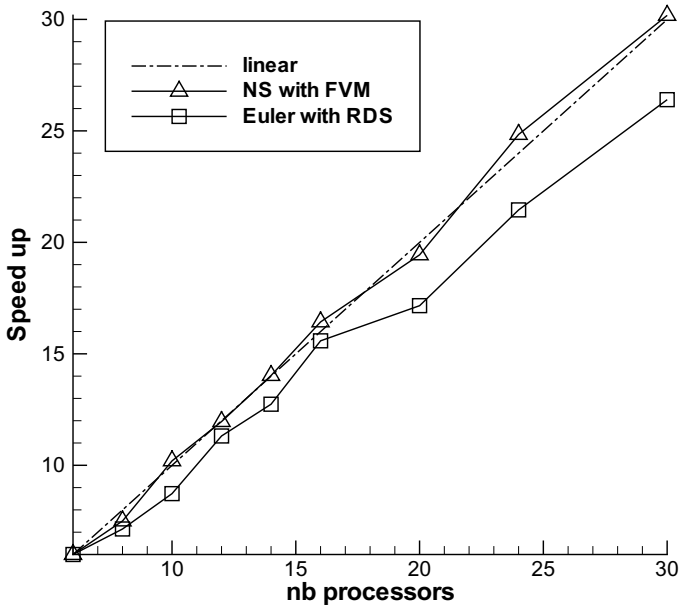

Fig. 3. Speed up results for two numerical simulations, one performed with FV and the other one with RD methods, and comparison with the ideal linear behaviour.

The performance of the numerical solvers implemented in COOLFluiD rely heavily upon DataHandles, since all mesh data are accessed through them. Since all accessor/mutator functions defined in DataHandles are inlinable (the profiler confirms this), efficiency is guaranteed. If we compare the direct usage of raw data and the usage through the DataHandle interface, the only price to pay is a single pointer indirection.

It's therefore not surprising that a good performance is achieved in our computations, as demonstrated by Fig. 3, where the speedup results of two simulations are presented. The first testcase is an implicit hypersonic Navier-Stokes computation on a Double Ellipsoid geometry (1972384 cells, 344315 nodes) with cell-centered Finite Volume (*FV*). The second one is an implicit transonic Euler simulation on a F15 geometry (3558667 cells, 621636 nodes) with cell-vertex Residual Distribution Schemes (*RDS*). All the computations have been performed on a cluster of Intel Pentium 4 machines with 2.8 or 3.4 GHz, 2 Gb of RAM, and MPI wall time has been used for the timing. The results show an almost perfect linear speed-up for the FV testcase and a close to linear speed-up for the RDS one. This different behaviour, in our opinion, can be explained by two reasons. First, the size of the overlap region, that includes all the vertex neighbors of the nodes on the partition boundary, is currently overestimated in a cell-vertex method like RDS. This leads to some useless additional work in both numerical computation and parallel communication that can impact on the overall efficiency, especially if we consider that in relatively small problems the overlap size can become comparable with the local size of the "updatable" mesh in each processor, as it is in our case. Second, the currently available space discretization algorithm for computing the residual and its perturbations on the partition boundaries is face-based in FV and can simply ignore useless faces, while in RDS it is cell-based and takes into account also vertices that should in fact be neglected. This clearly suggests that a further optimization of the RDS implementation is needed.

3. Implementation of physical models

Scientists and researchers in the field of Fluid Dynamics deal regularly with complex systems of PDEs expressing conservation laws that can be formulated as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F^C} + \nabla \cdot \mathbf{F^V} = \mathbf{S} \tag{1}$$

where **U** (state vector of unknown variables), F^c (convective fluxes), F^v (viscous fluxes), **S** (source or reaction term) depend on the chosen physical model. The latter can be seen as a composition of entities, e.g., transport coefficients, thermodynamic properties and quantities that contribute to define the mathematical description of some physical phenomena. It should be noticed that one can look at the same physics through different formulations of the corresponding equations, which involve the use of different sets of variables, transformations, or other adaptations tailored, e.g., towards a specific numerical method.

Moreover, in a framework for solving PDEs which supports different numerical techniques and run-time loading of components, the modules where algorithms are implemented should be as independent as possible from the modules dealing with the physics description. Therefore, when new physical models or numerical components are integrated, the need for modifications on the kernel, where most of the key abstractions are defined, should be avoided. Ideally, it should be possible to build more complex models by simply extending and reusing available ones, or solving the same equations by means of different numerical techniques, without requiring changes in the existing abstract interfaces and without establishing a high level direct coupling between physics and algorithms. Widely used OO CFD platforms like the ones described in [4,25] lack these last features, since the implementation of the physics is directly tangled to the numerical solver, in particular to the space discretization. This is surely a reasonable down-to-earth solution, it is probably efficient, but it does not promote enough reusability and interchangeability between physics and numerics. The `Perspective` pattern, that will now be described, proposes, in our opinion, a structural way for overcoming the above mentioned pitfalls and for facilitating dynamical incremental changes and code reuse.

3.1. Perspective pattern

Trying to define a single abstract interface for a generic physical model would be quite a demanding task and it would probably lead to a non maintainable solution, especially if we keep in mind our concern with run-time flexibility. In COOLFluiD, this hypothetical hard-to-define interface is therefore broken into a limited granularity of independent *Perspective* objects [13], each one offering a different view of the same physics, according to the specific needs of different numerical algorithms. To make an example, convective, diffusive, inertial, reaction or source terms of the equations can all be Perspectives with a certain abstract interface. Moreover, if one implements a new numerical scheme that needs to make use of an available physical model, but that requires something not foreseen a priori and, therefore, not offered by the available interfaces, a new Perspective can be created, without requiring a modification of the existing ones.

Figure 4 shows the Object Modeling Technique (OMT) [7] class diagram of a Perspective pattern applied to a generic physical model. The base `PhysicalModel` defines a very general abstract interface. `ConcreteModel` derives from it, implements the virtual methods of the parent class and defines another interface to which the concrete Perspective objects (and only those) are statically bound. This other interface offers data and functionalities which are typical of a certain physics, but invariant to all its possible Perspectives.

The resulting pattern lets the numerical client code make use of the physical model through an abstract layer, dynamically enlargeable if required, given by a number of Perspective objects, while all their collaborations with the ConcreteModel are completely hidden. The pattern reflects the composition-based *Adapter* described in [7], but with a single shared *Adaptee* object, namely `ConcreteModel`, and multiple abstract *Targets* (called Perspectives here), each one with a number of derived classes (*Adapters*). The fact that several objects may be defined to describe the same physics may look like a disadvantage, but, in our experience, improves reusability, allows much more easily to decouple physics from numerics and gives better support to a run-time *plug-in policy*.

Unlike in other OO CFD platforms [5,25] where no clear distinction between numerical algorithms and physical description is provided and where a close form of reliance (inheritance) binds the equation model and the scheme, in COOLFluiD, physics is completely independent and unaware of numerical methods (space and time discretizations, linear system solving, etc.). However, a design that is based on Perspective objects does not even prevent from having numerical objects, such as *Strategies* or *Commands* in 4.1, with static binding to the actual physics, e.g. in the case of some special schemes or boundary conditions. In the latter case, the use of Perspectives can still help to limit dependencies, improve reusability and avoid excessive sub-classing.

Some code examples are now presented in order to show the concrete applicability of the described pattern and its suitability to implement complex physical models.

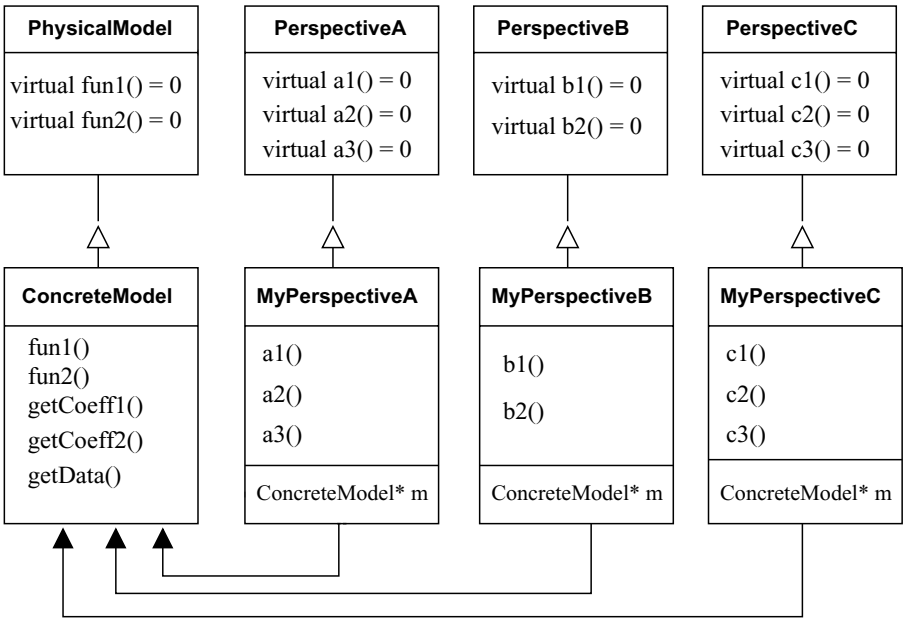

Fig. 4. Perspective pattern applied to a Physical Model.

3.1.1. Physical model object

At first, the interface of the base `PhysicalModel` is defined:

```
class PhysicalModel {
public:
  // enumerated variables used to define equation term types
  enum TermID {CONVECTION, DIFFUSION, REACTION, DISPERSION};

  PhysicalModel(string name); // constructor
  virtual ~PhysicalModel();   // destructor

  virtual void setUp() = 0;               //set up private data
  virtual int getDimension() const = 0;   //geometric dimension
  virtual int getNbEquations() const = 0; //number of equations

  // accessor-mutator function returning the BaseTerm
  // corresponding to the given TermID
  SafePtr<BaseTerm> getTerm(TermID t) {return m_tMap.find(t);}

protected:
  // registers the TermID and a pointer to the
  // corresponding polymorphic BaseTerm object
  void registerTerm(TermID t, BaseTerm* bt) {m_tMap.insert(t,bt);}
private:
  // mapping between TermID and a polymorphic physical term
  Map<TermID, SafePtr<BaseTerm> > m_tMap;
};
```

A polymorphic `BaseTerm` object and a corresponding enumerated variable are assigned to each independent physical term of the equation (convection, diffusion, reaction, dispersion, inertia, etc.) and they are registered in

an associative container, such as a `std::map`. BaseTerm manages a number of unidimensional arrays of floats (`RealVector`), where some physics-dependent data (thermodynamic quantities, transport coefficients, parameters useful for calculating fluxes, eigenvectors, eigenvalues etc.) are stored and continuously recomputed during the simulation. `SafePtr` is a wrapper class around a bare pointer that provides safe copy, prevents accidental deletion and does not imply ownership.

```
class BaseTerm {
public:
  // declarations of constructor, virtual destructor ...

  // sets the maximum number of physical data arrays:
  // this size must be set by the numerical solver
  // because it depends on the computational stencil
  void setNbDataArrays(CFuint maxSize);

  // get the size of each array of temporary physical data
  virtual CFuint getDataSize() const = 0;

  // finds the array of data  corresponding to the given
  // degree of freedom (state vector)
  SafePtr<RealVector> getPhysicalData(State* state);
};
```

In order not to affect the total memory requirements of the computation, the maximum number of the above mentioned physical data arrays is determined by the numerical method and scales with the number of degrees of freedom in one or more geometric entities (cells, faces, etc.), depending on the required computational stencil. To make an example, for a cell-vertex based algorithm (Finite Element, Residual Distribution, etc.), the total number of physical data arrays that are stored in BaseTerm is the maximum number of quadrature points in a cell. The size and the content of these arrays will be defined by the classes deriving from BaseTerm.

A template compositor object is defined for each possible combination of physical equation terms (Convection, Diffusion, ConvectionDiffusion, etc.). It derives from PhysicalModel and aggregates generic policies [1], i.e. subclasses of BaseTerms. Let's for instance consider a ConvectionDiffusion compositor:

```
template <class CT, class DT>
class ConvectionDiffusion : public PhysicalModel {
public:
  // constructor, virtual destructor,
  // overridden parent class pure virtual methods
protected:
  std::auto_ptr<CT> m_convTerm; // convective term
  std::auto_ptr<DT> m_diffTerm; // diffusive term
};
```

The constructor implementation is

```
template <class CT, class DT>
ConvectionDiffusion<CT,DT>::ConvectionDiffusion(string name) :
 PhysicalModel(name),
 m_convTerm(new CT(CT::getTermName())),
 m_diffTerm(new DT(DT::getTermName()))
{
  // register all the equation terms and their TypeIDs
  registerTerm(PhysicalModel::CONVECTION, m_convTerm.get());
  registerTerm(PhysicalModel::DIFFUSION, m_diffTerm.get());
}
```

The equation terms aggregated by the compositor are registered in the parent class and each one of them is made accessible polymorphically through the `getTerm()` method defined in PhysicalModel.

3.1.2. Navier-Stokes model

We consider now the case of a Navier-Stokes model. We implement the convective term `EulerTerm` and the diffusive term `NSTerm` as follows:

```
class EulerTerm : public BaseTerm {
public:
  // constructor, virtual destructor, etc.
  // enumerated variables: density, pressure, enthalpy, etc.
  enum {RHO=0, P=1, H=2, E=3, A=4, T=5,
        V=6, VX=7, VY=8, VZ=9};

  // number of the enumerated variables
  virtual CFuint getDataSize() const {return 10;}
  CFreal getMachInf() const {return m_machInf;}
  CFreal getTempRef() const {return m_tempRef;}
  CFreal getPressRef() const{return m_pRef;}
  static string getTermName() {return "Euler";}

  // get library to compute thermodynamic quantities
  SafePtr<TDLibrary> getThermodynamicLibrary() const
  {return m_tdlibrary;}

private:
  CFreal m_tempRef; //free stream temperature
  CFreal m_pRef;    //free stream pressure
  CFreal m_machInf; //free stream Mach number
  std::auto_ptr<TDLibrary> m_tdLibrary; // library
};

class NSTerm : public BaseTerm {
public:
  // constructor, virtual destructor, etc.
  // dynamic viscosity, thermal conductivity
  enum {MU=0, LAMBDA=1};

  // number of the enumerated variables
  virtual CFuint getDataSize() const {return 2;}
  CFreal getPrandtl() const {return m_Prandtl;}
  CFreal getReynoldsInf() const {return m_ReynoldsInf;}
  static string getTermName() {return "NS";}

  // get library to compute transport properties
  SafePtr<TPLibrary> getTransportPropertyLibrary() const
  {return m_tpLibrary;}

private:
  CFreal m_ReynoldsInf; // reference reynolds number
  CFreal m_Prandtl;     // Prandtl number
  std::auto_ptr<TPLibrary> m_tpLibrary; // library
};
```

Besides providing accessors methods for useful parameters and libraries computing physical quantities (i.e. thermodynamics or transport properties), EulerTerm and NSTerm define the size and, by means of enumerated variables, the entries for the arrays of physical data that are stored in `BaseTerm` and that have been previously introduced. It is now possible to define an instantiable `NavierStokesModel`, i.e. what is called `ConcreteModel` in the Perspective pattern:

```
class NavierStokesModel : public ConvectionDiffusion
                          <EulerTerm, NSTerm> {
public:
  // constructor, virtual destructor
  virtual CFuint getDimension() const;   //geometric dimension
  virtual CFuint getNbEquations() const; //number of equations
  virtual CFuint setUp() const; //set up data
};
```

NavierStokesModel can exploit the knowledge of both its concrete convective and diffusive terms to compute, for instance, adimensionalizing coefficients.

3.1.3. Variable sets

Numerical algorithms are not direct clients of the ConcreteModel. As explained previously, the latter is used through a layer of polymorphic Perspective objects, whose collaborations are statically bound to the ConcreteModel and therefore potentially very efficient. A variable set, `VarSet`, is a Perspective that decouples the usage of a physical model from the knowledge of the used variables. This, for instance, allows the client code to solve equations formulated in conservative variables, to perform intermediate algorithmic steps and to update in other ones (primitive, characteristic, etc.). Figure 5 shows two variable sets, `ConvectiveVarSet` and `DiffusiveVarSet`, from which `EulerVarSet` and `NavierStokesVarSet` respectively derive and whose class definitions follow:

```
class ConvectiveVarSet {
public:
  // constructor, virtual destructor, etc.

  virtual void setUp()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData(const State& state,
                               RealVector& data)=0;

  // set state vector starting from physical data
  virtual void setFromPhysicalData(const RealVector& data,
                                   State& state)=0;

  virtual void setJacobians(...)=0;   //set jacobians matrices
  virtual void splitJacob(...)=0;     //split the jacobian
  virtual void setEigenSystem(...)=0; //set the eigenvalues/vectors
  virtual void setEigenValues(...)=0; //set the eigenvalues
  virtual void computeFlux(...)=0;//compute the convective flux
};

class DiffusiveVarSet {
public:
  // constructor, virtual destructor, etc.
```

```
  virtual void setUp()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData(const State& state,
                               RealVector& data)=0;

  // set state vector starting from physical data
  virtual void setFromPhysicalData(const RealVector& data,
                                   State& state)=0;

  virtual void getWeakDiffMat(...)=0; //set the diffusive matrix
  virtual void computeFlux(...)=0; //compute the diffusive flux
};
```

Among all the methods declared in the above two class definitions, particular attention is due to `setPhysicalData()` and its dual `setFromPhysicalData()`: the former takes a given state vector (unknown variables) and sets the corresponding physical dependent data, whose entry pattern is defined by the the subclasses of BaseTerm; the latter does the opposite. In the case of the EulerTerm, for instance, if we assume that `State` holds primary variables (pressure, velocity components, temperature)

$$\vec{v} = [P, V_x, V_y, V_z, T] \tag{2}$$

`setPhysicalData()` will compute an array of physical quantities (density, pressure, total enthalpy, total internal energy, sound speed, temperature, velocity module and components)

$$\vec{w} = [RHO, P, H, E, A, T, V, V_x, V_y, V_z] \tag{3}$$

from $\vec{v}$ by means of thermodynamic relations. $\vec{w}$ will be then reused to compute eigenvalues, fluxes etc. A key point for the flexibility of the design is that each VarSet has acquaintance of the corresponding physical equation term (see Fig. 5), but not of the resulting ConcreteModel: this allows the developer to compose progressively more complex models with full reuse of the individual equation terms.

```
class EulerVarSet : public ConvectiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<EulerTerm> m_model;
};

class NavierStokesVarSet : public DiffusiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<NSTerm> m_model;
};
```

Some functionalities associated to a VarSet are independent on the variables in which one needs to work: advective fluxes, for instance, can always be computed once that physical data like pressure, enthalpy, velocity are known, because their formulation is always the same due to the conservation property.

However, as a counter example, the Euler equations can be written in conservative, symmetrizing, entropy, characteristic (...) variables and each one of this formulation defines different jacobian matrices for the convective fluxes. The VarSet abstraction is particularly useful in these more complex cases, since we can let variable-dependent

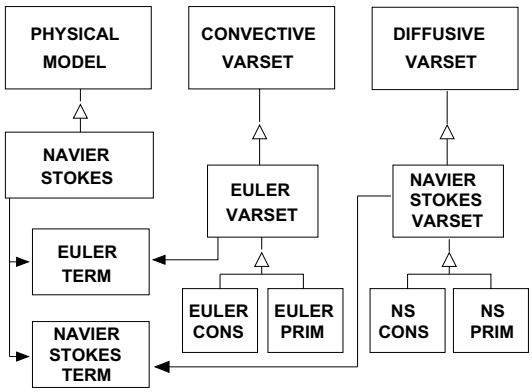

Fig. 5. Example of perspective pattern applied to the Navier-Stokes Model.

subclasses of EulerVarSet implement the jacobian matrices, according to the chosen formulation. Therefore you can have `EulerCons`(conservative), `EulerChar` (characteristic), `EulerSymm` (symmetrizing), etc.

It is helpful to have derived classes for the NavierStokesVarSet too, since, for instance, the transport properties can be computed differently in chemically reactive or non reactive flows, in local thermal equilibrium (LTE) or non equilibrium (NE). The method `computeFlux`() that calculates the diffusive flux is a template method [7] in NavierStokesVarSet and delegates the computation of the transport properties to subclasses, which have more specific knowledge. Some possible subclasses are `NavierStokesCons` (non reactive conservative variables), `NavierStokesLTEPvt` (pressure, velocity, temperature variables for LTE), `NavierStokesNERivt` (densities of chemical species, velocity, temperature variables for NE) etc. Another polymorphic Perspective is the `VariableTransformer`. This allows the client code to apply linear matrix transformations ($\partial U/\partial V$) between variables and to compute one set of variable from another (e.g. Euler primitive from Euler conservative and vice-versa). The combined use of VarSets and VariableTransformers gives the freedom to use different variables to update the solution, compute or distributed the residual, linearize jacobians, without requiring any modification in the client numerical algorithms that work with dynamically bound Perspective objects, completely unaware of the actual physics.

The support for variable independent algorithms which is offered by COOLFluiD represents an important feature that none of other well known CFD platforms such as [4,25] have.

3.1.4. Turbulent k-ε multi-species model

To verify the extensibility and soundness of the Perspective pattern, we can show an application to a more challenging case: a multi-species chemically reactive turbulent k-ε model. The compositor object is now a `ConvectionDiffusionReaction` and ConcreteModel will aggregate three terms: `EulerTurbKETerm`, `TurbKEDiffTerm` and `TurbKESourceTerm`. The following classes can therefore be defined:

```
class EulerTurbKETerm : public EulerTerm {
public:
  // turbulent kinetic  energy, viscous dissipation
  enum {K=11, EPS=12};

  // first species
  CFuint startSpecies() const {EulerTerm::getDataSize() + 2};

  // number of species
  CFuint getNbSpecies() const {return m_nbSpecies;}

  // total size of the physical data array, including
  // the entries declared by the parent class
```

```
  virtual CFuint getDataSize() const
  {return startSpecies() + m_nbSpecies;}

  // free stream turbulent kinetic energy
  CFreal getKref() const   {return m_kRef;}

  // free stream viscous dissipation
  CFreal getEpsRef() const {return m_epsRef;}
  static string getTermName() {return "EulerTurbKE";}

 // private data ...
};

class TurbKEDiffTerm : public NSTerm {
public:
  // turbulent dynamic viscosity
  enum {MUT=3};

  virtual CFuint getDataSize() const
  {return NSTerm::getDataSize() + 1;}

  // some coefficients
  CFreal getAlpha() const {return m_alpha;}
  CFreal getBeta() const {return m_beta;}
  static string getTermName() {return "TurbKEDiff";}
// private data
};

class TurbKESourceTerm : public BaseTerm {
public:
  // pairs of forward/backward reaction rates
  // corresponding to each chemical reaction are
  // stored in the physical data array
  virtual CFuint getDataSize() const
  {return m_nbReactions*2;}

  // some useful coefficients
  SafePtr<RealVector> getCoeffs() const {return m_coeffs;}
  static string getTermName() {return "TurbKESource";}
// private data
};

class TurbKEModel : public ConvectionDiffusionReaction
          <EulerTurbKETerm, TurbKEDiffTerm, TurbKESourceTerm> {
public:
  // constructor, virtual destructor
  virtual CFuint getDimension() const;
  virtual CFuint getNbEquations() const;
  virtual CFuint setUp() const;
};
```

It can be noticed that full reuse of existing terms is achieved. The data set is simply extended to accommodate the new model. The same reusability and flexibility applies also to the `VarSets` and `VariableTransformers`. In particular, `EulerKEVarSet` derives from EulerVarSet, acquaints EulerTurbKETerm, calls the parent method to compute the first five flux components and implements two additional ones for k and ε plus all the partial densities fluxes. Similarly, `TurbKEDiffVarSet` inherits from `NavierStokesVarSet`, acquaints TurbKEDiffTerm, and simply extends the implementation of the fluxes and computes the turbulent transport properties. Additionally, a `TurbKESourceVarSet`, that points to TurbKESourceTerm, and that implements a variable dependent behaviour for the reaction source term, needs also to be implemented.

4. Implementation of numerical methods

The solution of PDEs requires the implementation of different numerical techniques that deal with time or space discretizations, linear system solving, mesh adaptation algorithms, error estimation, mesh generation, etc.

In a typical OO design, e.g. like the ones proposed by [5,10,16,24], each one of these simulation steps is enclosed in separate object (or polymorphic hierarchies of objects) and different patterns are applied to make them all interact.

It would be advantageous to have just a single pattern, extraordinarily reusable, that allows the developer to encapsulate different numerical methods uniformly, and lets him/her focus more on the algorithm itself than on its already well-defined surrounding. This pattern should ease the integration of new algorithms, but also the implementation of different versions or parts of the same ones, while looking for optimal solutions and/or tuning for run-time performance.

One of the main achievements of COOLFluiD, as opposed to other similar OO environments, lies in having developed a uniform high level structural solution, potentially able to encapsulate and tackle efficiently a wide range of numerical algorithms: the compound Method-Command-Strategy pattern.

4.1. Method-Command-Strategy pattern

The *Method-Command-Strategy* (MCS) pattern [13,17] provides a uniform way to implement numerical algorithms and to compose them according to the specific needs. As sketched in Fig. 6, `BaseMethod` defines an abstract interface for a specific type of algorithm (e.g. MeshCreator, SpaceMethod, ConvergenceMethod, ErrorEstimator, LinearSystemSolver). `ConcreteMethod` implements the virtual functions of the corresponding parent class by delegating tasks to ad-hoc `Commands` [1,7] that share a tuple, `ConcreteMethodData`, aggregating multiple polymorphic receivers (`Strategies`). Three levels of abstraction and flexibility can be identified in this pattern: `BaseMethod`, `Command` and `Strategies` can all be overridden, allowing one to implement the same task, at the corresponding level, in different ways. This kind of behavioral modularity allows the developers to easily re-implement or tune components (Methods, Commands or Strategies), and gives them the freedom to move code from one layer to another, according to convenience, taste or profiling-driven indications. The fast-path code, critical for the overall performance, can be wrapped inside Commands or Strategies and it can be substituted with more efficient implementations without implying changes in the upper layer.

Moreover, while freedom is left to define the abstract interface of a new polymorphic BaseMethod or Strategy, the interface of a Command consists of only three actions:

```
virtual void setup();   // setup private data
virtual void unsetup(); // unsetup private data
virtual void execute(); // execute the action
```

In COOLFluiD, managing the collaboration between different numerical methods is eased by the fact that Commands can create their own local or distributed data and share them with other Commands defined within other Methods, by making use of the `DataStorage` facility, whose details are presented in Section 2.1.

Moreover, *Perspective* objects, as described in 3.1, can be used within the MCS pattern, at the Strategy-level, as part of the `ConcreteMethodData`, in order to bind a numerical algorithm to the physics polymorphically.

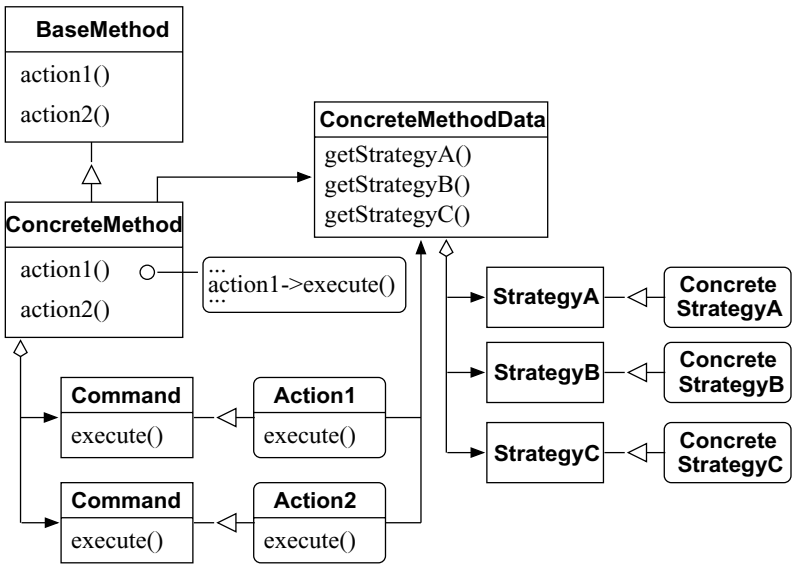

Fig. 6. OMT diagram of a MCS pattern.

This collaboration between MCS and Perspective patterns is an effective solution to minimize the coupling between physics and numerics. It makes possible, for instance, to employ a certain space method to discretize equations related to different physical models, but also to apply different space methods that require completely different data-structures to discretize a single set of equations.

Interchangeability of Methods, Commands and Strategies can be facilitated and maximized by making them *self-registering* and *self-configurable* objects [3,13]: this gives users and developers full control on each of them.

4.1.1. Self-registering objects

The self-registration technique [3,13] automatizes the creation of polymorphic objects and reduces implementation and compilation dependencies. A generic `ConcreteObj` of polymorphic type `BaseObj` can be registered by simply instantiating the corresponding `ObjectProvider` in the implementation file:

```
ObjectProvider<BaseObj, ConcreteObj> myProvider("objName");
```

The string *"objName"*, accepted by the provider constructor, can then be used as a key to ask a singleton template `Factory` [7] [13] to create the corresponding polymorphic object:

```
BaseObj* obj = Factory<BaseObj>::getProvider("objName")->create();
```

4.1.2. Self-configurable objects

In COOLFluiD, objects can be self-configurable [13], i.e. they can create and set their own data. An object is made self configurable, by deriving it from a parent class `ConfigObject` and by adding a call to

```
addConfigOption("OptionKey","option description", &configData);
```

in its constructor for each configurable data member *configData*. In particular, `OptionKey` is the configuration key string, used to map the value of *configData*. This technique allows the user to input whatever kind of data (including analytical functions) from file, environmental variables or command line options. We consider, as a basic example, what could appear in a configuration file for a concrete SpaceMethod object:

```
SpaceMethod = MySM          # name of the concrete SpaceMethod
MySM.SetUpCom = MySetUp     # name of the setup Command
MySM.Data.StrategyA = MyA   # name of the strategy of type A
MySM.Data.StrategyB = MyB   # name of the strategy of type B
```

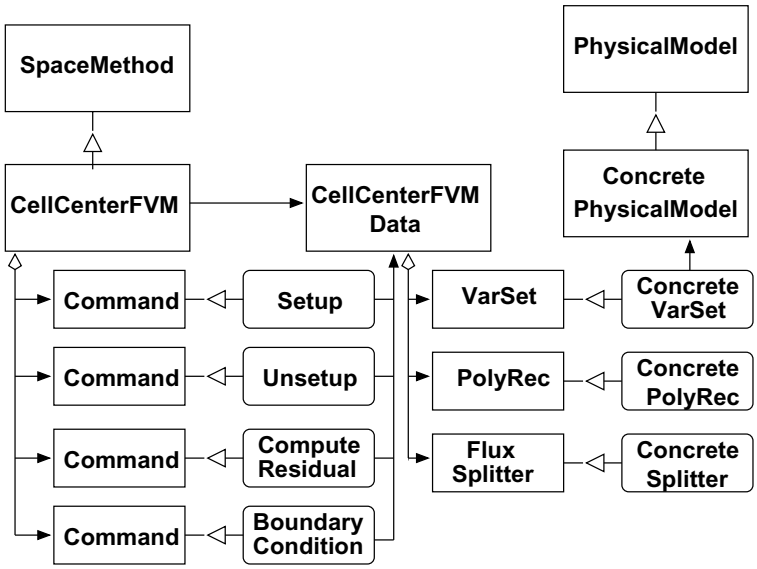

Fig. 7. MCS pattern applied to a Finite Volume module.

MySM is the self-configuration value for *SpaceMethod*, which is the configuration key for the homonymous polymorphic object. *MySM* is also the self-registration key for the concrete space method class, `MySpaceMethod`, that will then be instantiated and will configure itself with the specified setup Command, named *MySetUp*, and data. The latter includes two polymorphic strategies, whose selected instances are called *MyA* and *MyB*.

A deeper analysis of the MCS pattern and more implementation details are provided in [17]. Next we consider some possible applications.

4.1.3. Example of SpaceMethod: FiniteVolume

We define the interface of a `SpaceMethod`, that takes care of the spatial discretization of the given set of PDEs, according to a specified numerical scheme, on a chosen mesh:

```
class SpaceMethod : public Method {
public:
  SpaceMethod(string name);       // constructor
  virtual ~SpaceMethod();         // virtual destructor
  virtual void setMethod() = 0;   // setup the method
  virtual void unsetMethod() = 0; // unsetup the method

  // initialize the solution
  virtual void initializeSolution(CFbool isRestart) = 0;

  // compute the space part of the residual and jacobian matrix
  virtual void computeSpaceResidual(CFreal factor=1.0) = 0;

  // compute the time dependent part of the residual and
  // jacobian matrix
  virtual void computeTimeResidual(CFreal factor=1.0) = 0;

  virtual void applyBC() = 0; // apply boundary conditions
};
```

As shown in the sample code above, SpaceMethod inherits from a non instantiable `Method` object that provides configuration functionalities meant to be reused by all its children, namely all the possible `BaseMethods` in Fig. 6. Figure 7 shows a simplified class diagram of the cell center `Finite Volume` (FV) module. `CellCenterFVM` is a concrete instance of SpaceMethod. Specific Commands, `CellCenterFVMCom`, are associated to actions like *setup* and *unsetup* (creation and destruction of data needed by the employed scheme), application of *boundary conditions*, computation of the residual and jacobian contributions to the system matrix, if required by implicit time stepping:

```
class CellCenterFVM : public SpaceMethod {
public:
  typedef SelfRegistPtr<Command<CellCenterFVMData> > FVMCom;
  // constructor, destructor, overridden virtual functions

private:
  // data to share between FVMCom commands
  std::auto_ptr<CellCenterFVMData> m_data;

  std::pair<FVMCom,string> m_setup;   // setup Command
  std::pair<FVMCom,string> m_unsetup; // unsetup Command

  // Commands computing the residual/jacobian
  std::pair<FVMCom,string> m_computeSpaceRHS; // space part
  std::pair<FVMCom,string> m_computeTimeRHS;  // time part

  // Commands that initialize the solution in the domain
  std::vector<FVMCom> m_inits;
  std::vector<string> m_initsStr; // init Commands names

  // Commands that computes the boundary conditions (bc)
  std::vector<FVMCom> m_bcs;
  std::vector<string> m_bcsStr; // bc Commands names
};
```

All self-registering polymorphic objects, including Commands, are aggregated by `SelfRegistPtrs`, i.e. smart pointers with intrusive reference counting [1] that keep ownership on them. In order to take full profit of self-registration and self-configuration features, all `ConcreteMethods`, including CellCenterFVM, hold the Command names (second entry in the *std*::*pair* tuples), which are used as keys for the polymorphic creation and configuration of the Commands themselves. Let's consider the constructor of CellCenterFVM:

```
CellCenterFVM::CellCenterFVM(string name) :
  SpaceMethod(name), m_data(new CellCenterFVMData())
{
  m_setupStr = "StdSetup"; // default name
  addConfigOption("SetupCom","Setup",&m_setup.second);

  m_unSetupStr = "StdUnSetup"; // default name
  addConfigOption("UnSetupCom","UnSetup",&m_unSetup.second);

  m_computeSpaceRHSStr = "FVMCCRhs"; // default name
  addConfigOption("ComputeRHS", "Compute space residual",
                  &m_computeSpaceRHS.second);
  // ...
}
```

All the Command names are set to a default that can be overridden by the user in the COOLFluiD input file: these will cause the object requested by the user to be instantiated, if available. Let's analyze the following fragment of an input configuration file:

```
SpaceMethod = CellCenterFVM
CellCenterFVM.SetupCom = LeastSquareP1Setup
CellCenterFVM.UnSetupCom = LeastSquareP1UnSetup
CellCenterFVM.ComputeRHS = NumJacob
```

This asks to create the SpaceMethod corresponding to the name CellCenterFVM and to select *setup* and *unsetup* Commands specific for a second order scheme based on least square reconstruction, which requires the allocation of many more data (cell limiters, cell gradients, weights etc.) than a first order one. Likewise, the Command with name `NumJacob` will be used to compute the residual and jacobian contributions instead of the default one, called `FVMCCRhs`.

All FVMComs are parameterized with a policy class [1], `CellCenterFVMData`, which groups together all the Strategy objects needed by the Commands to fulfill their job: `FluxSplitter`, the flux splitting scheme, `PolyRec`, the polynomial reconstructor, some *Perspectives*, i.e. `VarSets` and `VariableTransformers` (see 3.1), that provide the binding to the physics.

```
class CellCenterFVMData : public ConfigObject {
public:
  //constructor, destructor, configuration functions

  // convective, diffusive flux, source term computers
  SafePtr<FluxSplitter> getFluxSplitter() const;
  SafePtr<DiffusiveFluxComputer> getDiffFluxComputer() const;
  SafePtr<ComputeSourceTerm> getSourceTermComputer() const;

  // solution and update convective variable sets
  SafePtr<ConvectiveVarSet> getSolutionVar() const;
  SafePtr<ConvectiveVarSet> getUpdateVar() const;

  // update diffusive var set
  SafePtr<DiffusiveVarSet> getDiffusiveVar() const;

  // polynomial reconstructor
  SafePtr<PolyReconstructor> getPolyReconstructor() const;

  // vectorial transformer from update to solution variables
  SafePtr<VarSetTransformer> getUpdateToSolutionVecTrans() const;

  // other accessors/mutators ...
private:
 std::pair<SelfRegistPtr<FluxSplitter>, string>  m_fluSplitter;
 std::pair<SelfRegistPtr<PolyReconstructor>, string>  m_polyRec;
 // the same for all the other objects ...
};
```

Since CellCenterFVMData is also a self-configurable object, this implies that the user can select the concrete Strategies by name at run-time (e.g. *Roe*, *AUSM*, *LaxFriedrichs* as FluxSplitter, *Constant* or *LeastSquare* as PolyRecostructor, etc.), while the developer can implement and register new ones without needing to modify the client code.

Other subclasses of SpaceMethod, such as `Finite Element` (FE) or `Residual Distribution`(RD), are implemented in a similar way.

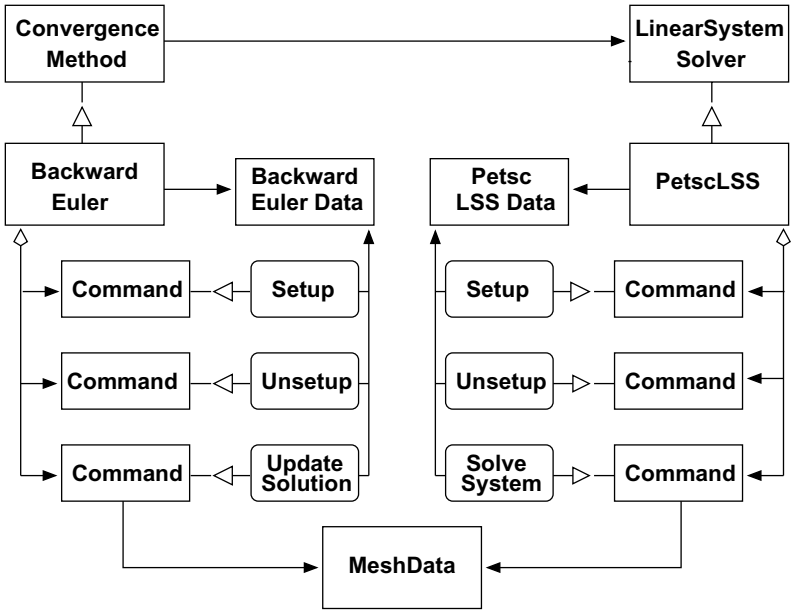

Fig. 8. MCS pattern applied to two interacting modules: a Backward Euler convergence method and a Petsc linear system solver.

4.1.4. Collaboration between two methods: ConvergenceMethod and LinearSystemSolver

We consider now an example of interaction between two different numerical methods. Figure 8 shows the collaboration between two abstract methods: `ConvergenceMethod`, responsible of the iterative procedure, and `LinearSystemSolver`. In this case, `BackwardEuler`, an implicit convergence method, delegates polymorphically the solution of the resulting linear system to `PetscLSS`, which interfaces the `PETSc` library [2]. `BackwardEuler` makes use of Commands for the setup, unsetup and solution update, while `PetscLSS` let Commands implement the setup, unsetup and solution of the linear system.

The class definition of ConvergenceMethod is the following:

```
class ConvergenceMethod : public Method {
public:
  //constructor, virtual destructor, configuration methods

  virtual void takeStep() = 0;    // Take one timestep
  virtual void setMethod() = 0;   // Sets up private data
  virtual void unsetMethod() = 0; // Clears up private data

  // Sets collaborator methods (SpaceMethod, LinearSystemSolver)
  void setCollaborator(MultiMethodHandle<SpaceMethod> spaceMtd);
  void setCollaborator(MultiMethodHandle<LinearSystemSolver> lss);
protected:
  // Synchronizes the states and computes the norm of the residual
  void synchAndComputeRes(CFbool computeResidual);

protected: //data
  // Space Method used to compute spatial discretization
  MultiMethodHandle<SpaceMethod> m_spaceMtd;

  // LinearSystemSolver used to solve the linear system (if any)
```

```
  MultiMethodHandle<LinearSystemSolver> m_lss;
};
```

In the code above, `MultiMethodHandle` is a lightweight proxy object that hides the knowledge of multiplicity: it controls the access to one or more underlying Methods with the same polymorphic type and dispatches specified actions on all of them sequentially, similarly to what a *std::for_each* function would do [20]. The purpose of accessing Method objects through MultiMethodHandles is to offer transparent support for weakly coupled simulations, where, in the same process, two or more different linear systems are assembled by one or more SpaceMethods and must be solved one after the other.

SpaceMethod (SM) and LinearSystemSolver (LSS) are the collaborator Methods: a concrete ConvergenceMethod uses them polymorphically via MultiMethodHandles, without knowledge of their concrete type or their actual number.

As a result, concrete collaborator Methods are completely interchangeable with other ones with the same polymorphic type.

When running in parallel, in the function `synchAndComputeRes`() the global storage of state vectors and nodal coordinates are accessed via `DataHandle` and asked to synchronize the underlying parallel array:

```
void ConvergenceMethod::synchAndComputeRes(CFbool computeResidual)
{
  // handle to the global storage of states
  DataHandle<GLOBAL,CFreal> statedata = MeshData::getInstance()->
    getGlobalData()->getData<CFreal>("statedata");

  // handle to the global storage of nodes
  DataHandle<GLOBAL,CFreal> nodedata = MeshData::getInstance()->
    getGlobalData()->getData<CFreal>("nodedata");

  // after each update phase states and nodes
  // have to be synchronized
  statedata->beginSync();
  nodedata->beginSync();

  if (computeResidual) {...}  // computation of the residual

  statedata->endSync();
  nodedata->endSync();
}
```

In a parallel simulation, the norm of the residual is calculated with a collective operation, between the beginning and the end of the synchronization process, in order to overlap communication and computation and maximize the efficiency of the operation. When running serially, the implementation of `synchAndComputeRes`() remains unchanged, but in fact the calls to `beginSync`() and `endSync`() have no effect.

We consider now BackwardEuler, a subclass of the ConvergenceMethod:

```
class BackwardEuler : public ConvergenceMethod {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<BwdEulerData> > BwdEulerCom;

  void takeStep();    // Take one timestep
  void setMethod();   // Sets up private data
  void unsetMethod(); // Clears up private data
```

```
private:
  //data shared by BwdEuler Commands
  std::auto_ptr<BwdEulerData> m_data;
  std::pair<BwdEulerCom,string> m_setup;     // set up
  std::pair<BwdEulerCom,string> m_unSetup;   // unsetup
  std::pair<BwdEulerCom,string> m_updateSol; // update solution
};
```

A different Command and its name are associated to each one of the pure virtual functions declared by the parent ConvergenceMethod. We report here the implementation of `takeStep()`, where the polymorphic usage of the collaborators (SM and LSS) and of the solution updating Command is shown.

```
void BwdEuler::takeStep()
{
  // compute residual and jacobian contributions for
  // the spatial and time dependent term of the equations
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeSpaceResidual));
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeTimeResidual));

  // solve the resulting linear system
  m_lss.apply(mem_fun(&LinearSystemSolver::solveSys));

  m_updateSol.first->execute(); // update the solution

  // synchronize nodes and states, compute the intermediate residual
  ConvergenceMethod::synchAndComputeRes(true);
}
```

The code is readable, concise, independent from the actual type and number of SM or LSS, extremely flexible, since each part of the algorithm (Commands or Methods) can be replaced at run-time without any performance overhead. In fact, the frequency of virtual calls in question is exceptionally low and, therefore, does not have an impact on the run-time speed.

The class definitions of the parent LSS and of its subclass PetscLSS are the following:

```
class LinearSystemSolver : public Method {
public:
  //constructor, virtual destructor, configuration methods
  virtual void solveSys() = 0;    // solve the linear system
  virtual void setMethod() = 0;   // setup private data
  virtual void unsetMethod() = 0; // clear up private data

  // create a block accumulator with ad-hoc internal storage
  virtual BlockAccumulator* createBlockAccumulator
  (int nbRows, int nbCols, int subBlockSize) const = 0;

  // accessor/mutator for local to global LSS index mapping
  SafePtr<LSSIdxMapping> getLocalToGlobalMapping()
  {return &_localToGlobal;}

  // get the system matrix
  virtual SafePtr<LSSMatrix> getMatrix() const = 0;
private:
  // idx mapping from local to LSS global
  LSSIdxMapping m_localToGlobal;
};
```

```
class PetscLSS : public LinearSystemSolver {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<PetscLSSData> > PetscLSSCom;

  void setMethod();   // setup private data
  void unsetMethod(); // clears up private data
  void solveSys();    // solve the linear system

  // create a block accumulator with ad-hoc internal storage
  BlockAccumulator* createBlockAccumulator() const;

  // get the system matrix
  SafePtr<LSSMatrix> getMatrix() const
  {return &m_data->getMatrix();}

private:
  ///The data to share between PetscLSSCom Commands
  std::auto_ptr<PetscLSSData> m_data;
  std::pair<PetscLSSCom,string> m_setup;    // setup Command
  std::pair<PetscLSSCom,string> m_unSetup;  // unsetup Command
  std::pair<PetscLSSCom,string> m_solveSys; // solver Command
};
```

PetscLSS delegates tasks to specific Commands, PetscLSSCom, sharing some data (`PetscLSSData`), such as references to the (parallel) Petsc matrix and the (parallel) Petsc vectors involved in the solution of the linear system:

```
class PetscLSSData {
public:
  //constructor, destructor, configuration methods

  PetscVector& getSolVec() {return m_xVec;} //Petsc solution array
  PetscVector& getRhsVec() {return m_bVec;} //Petsc rhs array
  PetscMatrix& getMatrix() {return m_aMat;} //Petsc matrix
  PC& getPreconditioner() {return m_pc;} //Petsc preconditioner
  KSP& getKSP() {return m_ksp;} //Petsc Krylov solver

  // accessors to various Petsc parameters, private data, etc.
};
```

Only the PetscLSSCom can make direct use of PETSc [2] objects like `PC` or `KSP`, that are aggregated by PetscLSSData. The knowledge of a specific LSS, PetscLSS, in this case, is not assumed anywhere in the numerical modules, thanks to the use of abstractions such as `LSSMatrix`, `BlockAccumulator` and `LSSIdxMapping`. `LSSMatrix` is the parent system matrix from which PetscMatrix derives. `BlockAccumulator` bundles blocks of values to be inserted in the matrix. `LSSIdxMapping` stores a mapping from the local numbering to an optimal LSS-dependent global one.

As represented in Fig. 8, all the involved Commands have acquaintance of MeshData, which provides access to DataStorage and DataHandles, and can therefore use and modify bulk data, qualified by name and type, as explained in 2.1. In other words, while, on one hand, each ConcreteMethodData, such as PetscLSSData and BackwardEulerData, allows intra-Method sharing of ConcreteMethod-dependent data among Commands, on the other hand, MeshData is the vehicle for inter-Method data exchange.

The following sample code should help clarifying the last statement in the case of two Commands, namely `UpdateSol` in BackwardEuler and `SolveSys` in PetscLSS:

```
void UpdateSol::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MeshData::getInstance()->
    getLocalData()->getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MeshData::getInstance()->
    getLocalData()->getData<CFreal>("rhs");

  // compute the solution update ...
}

void SolveSys::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MeshData::getInstance()->
    getLocalData()->getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MeshData::getInstance()->
    getLocalData()->getData<CFreal>("rhs");

  // get the method data shared by all PetscLSS Commands
  PetscMatrix& mat = getDataPtr().getMatrix();
  PetscVector& rhsVec = getDataPtr().getRhsVector();
  PetscVector& solVec = getDataPtr().getSolVector();
  KSP& ksp = getDataPtr().getKSP();

  // perform final assembly and ask PETSc to solve the system
}
```

Data stored in MeshData are allowed to cross the Method scope and can be used in Commands belonging to different Methods. The keys for the data exchange are the storage name and type, that must both match in all the method Commands that need the same data.

In parallel simulations, nothing changes, since the Commands implementing numerical algorithms work only with `LOCAL` data, as they would do in a serial run. Functions that demand access to the `GLOBAL` storage and perform some parallel action, like the above mentioned `synchAndComputeRes`(), are exceptional.

Furthermore, the example of the collaboration between ConvergenceMethod and LSS demonstrates the suitability of the MCS pattern to interface existing libraries, PETSc in this case, without exposing any detail of their actual implementation to their clients.

The *Trilinos* package [19] has also been successfully integrated in COOLFluiD by means of the MCS pattern. While the class definitions of the corresponding concrete linear system solver Method and the related Commands are basically similar to the Petsc's ones, the interface of TrilinosLSSData is defined as follows:

```
class TrilinosLSSData {
public:
  // map for the individual IDs of the unknowns
  Epetra_Map* getEpetraMap() {return m_map;}

  TrilinosVector* getSolVec() {return &m_xVec;} //solution vector
  TrilinosVector* getRhsVec() {return &m_bVec;} //rhs vector
  TrilinosMatrix* getMatrix() {return &m_aMat;} //system matrix
  AztecOO* getKSP() {return &_ksp;} // Aztec Krylov solver
```

```
  // various options and parameters to control convergence
  // and tune the solver to satisfy the user needs ...
 };
```

To summarize, the application of the MCS pattern helps to encapsulate not only each numerical algorithm but also its collaborations with other algorithms. This contributes to enforce the multi-component-oriented character of the COOLFluiD framework, where each component can be replaced by another one with the same polymorphic type, without affecting the client code.

5. Conclusions

In the available literature related to OO simulation of PDEs, to the knowledge of the authors, it's difficult to find design solutions that can offer, at least in principle, a flexibility in the serial and parallel data handling, a run-time interchangeability between physics and numerics and a structural support to encapsulate generic numerical algorithms in a flexible and reusable way, comparable with the ones presented in this article. As a matter of fact, the proposed design and implementation ideas don't necessarily need to be employed within COOLFluiD, but they can be reused independently and in other scientific computing contexts, even unrelated to the solution of PDEs.

The flexible and reusable design solutions presented in this paper have allowed COOLFluiD to enlarge its target applications beyond the scope of solely Computational Fluid Dynamics. Several components have already been integrated in the framework: explicit (Runge-Kutta) and implicit (Newton, Crank-Nicholson, Three Point Backward) time stepping, different spatial discretizations (FV, RD, Space-Time RD, FE), different physical models (Euler, Navier-Stokes, ideal Magneto Hydro Dynamics, Linear Elasticity, Heat transfer, etc.), different linear system solvers wrappers (PETSc [2], Trilinos [19]), a parallel flexible data-structure supporting the use of hybrid meshes, etc.

The implementation of many other functionalities is underway: Aero-Thermo-Chemical models, incompressible plasma flows, error estimation, mesh movement and adaptation, loosely coupled fluid-structure interaction.

Acknowledgments

T. Quintino acknowledges the financial support of Fundaç ão para a Ciência e Tecnologia, under the fellowship SFRH/BD/9080/2002.

References

[1] A. Alexandrescu, Modern C++ design, Addison Wesley, 2001.
[2] Argonne National Laboratory, PETSc: Portable, Extensible Toolkit for Scientific Computation, http://www-unix.mcs.anl.gov/petsc, 2004.
[3] J. Beveridge, Self-Registering Objects in C++, Dr. Dobbś Journal, 8/1998.
[4] A.M. Bruaset, E.J. Holm and H.P. Langtangen, Increasing the Efficiency and Reliability of Software Development for Systems of PDEs, in: *Modern Software Tools for Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen, eds, Birkhäuser, 1997.
[5] A.M. Bruaset and H.P. Langtangen, *A Comprehensive Set of Tools for Solving Partial Differential Equations*; Diffpack, Numerical Methods and Software Tools in Industrial Mathematics, Birkhuser, 1997.
[6] D. Bulka and D. Mayhews, Efficient C++. Performance programming techniques, Addison Wesley, 2000.
[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[8] A. Geist, A. Beguelin, J. Dongarra et al., *PVM: Parallel Virtual Machine: a Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1995.
[9] J. Hardtlein, A. Linke and C. Pflaum, Fast Expression Templates: Object-Oriented High Performance Computing, Computational Science – ICCS 2005, LNCS 3514, (Vol. 2), Springer-Verlag, V.S. Sunderam, G.D. van Albada, P.M.A. Sloot and J.J. Dongarra, eds, Atlanta, 2005, pp. 1053–1063.
[10] B. Henshaw et al., Object-Oriented Tools for Solving PDEs in Complex Geometries, http://www.llnl.gov/casc/Overture, 2004.
[11] D. Kimpe, S. Vandewalle and S. Poedts, EVector: an Efficient Vector Implementation, POOSC'05 Workshop Notes, Glasgow, 2005 (to be published).

[12] D. Kimpe, A. Lani, T. Quintino, S. Poedts and S. Vandewalle, The COOLFluiD Parallel Architecture, in: *Proc. 12th European Parallel Virtual Machine and Message Passing Interface Conference*, B. Di Martino, D. Kranzlmller and J.J. Dongarra, eds, Sorrento, 2005.
[13] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle and S. Poedts, The COOLFluiD Framework: Design Solutions for High-Performance Object Oriented Scientific Computing Software, Computational Science – ICCS 2005, LNCS 3514, Vol. 1, Springer-Verlag, V.S. Sunderam, G.D. van Albada, P.M.A. Sloot and J.J. Dongarra, eds, Atlanta, 2005, pp. 281–286.
[14] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org.
[15] S. Meyers, Effective C++ CD. 85 specific ways to improve your programs and designs, Addison Wesley, 1999.
[16] G. Nelissen and P.F. Vankeirsbilck, Electrochemical Modelling and Software Genericity, in: *Modern Software Tools for Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen, eds, Birkhäuser, 1997.
[17] T. Quintino and A. Lani, *Method-Command-Strategy Pattern: A Multi-Component Solution for High-Performance Scientific Computing*, POOSC'05 Workshop Notes, Glasgow, 2005 (to be published).
[18] G. Karypis, K. DeRonne et al., ParMETIS: Parallel Mesh Partitioning, http://www-users.cs.umn.edu/˜karypis/metis/parmetis, 2003.
[19] Sandia National Laboratories, The Trilinos Project, http://software.sandia. gov/trilinos, 2004.
[20] B. Stroustrup, The C++ programming language, Addison Wesley, 2001.
[21] H. Sutter, *More exceptional C++*, 40 new engineering puzzles, programming problems, and solutions, Addison Wesley, 2002.
[22] D. Vandevoorde and N.M. Josuttis, *C++ Templates*, The complete guide, Addison Wesley, 2003.
[23] T. Veldhuizen, Expression Templates, C++ Report 7, 1995, 26–31.
[24] R. Vilsmeier, T. Stoye and N. Stuntz, *A C++ Library for Finite Volume Simulations*, http://www.vug.uni-duisburg.de/MOUSE, 2002.
[25] H. Weller et al., OpenFOAM: The Open Source CFD Toolbox, http:// www.opencfd.co.uk/openfoam, 2004.