Adding Aspect-Oriented Concepts to the High-Performance Component Model of SBASCO

Manuel Díaz, Sergio Romero, Bartolomé Rubio, Enrique Soler and Jose María Troya
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga
29071 Málaga, SPAIN
{*mdr,sromero,tolo,esc,troya*}*@lcc.uma.es*

Abstract

SBASCO provides a new programming model for parallel and distributed numerical applications which exploits the combination of software components and skeletons. This paper presents an extension to both the model and implementation of SBASCO, so that the notion of aspect is applied in conjunction with the original paradigms. The objective is to achieve a higher level of modularity and reuse in parallel scientific codes and applications. Our aspects are managed as components which implement the (sequential or parallel) cross-cutting functionality. Aspects interact with the base code by means of connectors that express the cross-cutting nature of the target concerns. The way in which both aspect weaving and advice code execution are managed is critical for preserving the performance of applications. An implementation of the abstractions for distributed memory parallel systems based on MPI is discussed.

1. Introduction

Parallel scientific applications are characterized by the use of low-level abstractions mechanisms to achieve accurate performance control. Some examples are the use of parallel libraries (e.g. MPI, p-threads, PVM) from a host programming language. As the application developer is aware of the cost of the communication primitives on specific target platforms, she/he can design efficient algorithms.

Despite parallel programming techniques having greatly evolved in recent years, modern paradigms of software engineering are hardly ever applied to High-Performance Computing (HPC). One example is that of Aspect-Oriented Programming (AOP) [8], the solution for modular and centralized management of cross-cutting concerns which aims at achieving simpler code, easier to develop and maintain, and which has greater potential for reuse. The proposals that use aspects for the modularization of high-performance concerns are often based on AspectJ [13], a general purpose language extension to Java for AOP. Although the amount of work in this area is relatively limited, some significant efforts can be found in [11][12][16].

In SBASCO [5][6], the development of parallel and distributed numerical applications is carried out combining components and parallel skeletons. On the one hand, software components [10] represent the key for the construction of extensible and adaptable systems. On the other hand, skeletons [14] (also known as patterns) are generic and reusable parallelism forms which can be used to establish the parallel structure of an application in a high-level and elegant way.

Other component-based approaches oriented to HPC applications are CCA [1], ASSIST [18] and PaCO [15]. With respect to skeletal programming, eSkel [2], Muskel [4] and LLC [7] are interesting proposals. With the exception of ASSIST, none of these technologies combine the notion of skeleton with software component, as is done in SBASCO.

Although SBASCO represents an improvement on classical development styles, designs based on this model may suffer from a lack of modularity if the application concerns spread over (i.e. cross-cut) the set of participant components. In order to ease the level of code-tangling as well as achieving finer grain application designs, this work defines new concepts on top of SBASCO to allow aspects to be used in conjunction with the previously used paradigms.

In the new approach, aspects represent well-modularized cross-cutting concerns which are encapsulated into components. In this scenario, aspect weaving becomes component composition, and interactions between aspects and base code are defined in terms of invocations on the component interface. Extra-functional properties, typically managed by means of aspects, can be plugged into the application more easily (or unplugged if necessary). Our aspects are not invasive in the sense that they are not allowed to alter the implementation of the base components. As a consequence, the main principles of component-based development are preserved in this hybrid model. The base components are dealt with as black-boxes that will execute aspect code at different points of the control flow. The base components are aware of the potential effects caused by aspect execution, as the former specify the set of methods aspects can invoke to access the application internal state.

The nature of aspects in our application domain can be

very diverse: from specific features of the numerical methods (e.g. convergence, linear solvers) to generic concerns such as parallel I/O, dynamic processor re-mapping, persistence of the computation state, and so on. The goal pursued is a clear separation of concerns which allows the mathematical model and core functions to be expressed better.

The high-level abstractions that extend SBASCO need to be supported by an efficient implementation. Actually, the mechanisms proposed for aspect execution are added to the current system implementation in a way that the performance is not compromised. This paper not only describes the language and model extension, but it also explores a way to execute the new applications (which include the layer of aspects) with minimum overhead.

This paper is organized as follows. In Section 2 we describe both, the original model of SBASCO and the aspect-oriented extension proposed. In Section 3, a system implementation which is based on the message passing interface is discussed. An example that shows the use of aspects is described in Section 4. The paper finishes with some conclusions.

2. Aspect-Oriented Extension to SBASCO

This section provides a description of SBASCO (Skeleton-BAsed Scientific COmponents). Then, the new abstractions which allow the developer to include aspects in her/his applications are defined.

2.1. The Component Model of SBASCO

SBASCO provides a new component model focussed on the efficient development of scientific software.

The different (parallel or sequential) tasks that solve the numerical problem are encapsulated into the so-called Scientific Components (SCs). A SC represents an application composition unit which can be executed on a set of processors.

Communication between SCs is based on a data-flow style governed by two primitives, called `get_data()` and `put_data()`. The SC interface is the element used to express the input and output arguments of the component. In addition, the SC interface also describes information about the argument data distribution and component processor layout. This type of information, available at the component interface level, is the key to implementing efficient point-to-point data communication between SCs.

SBASCO defines a family of parallel skeletons which can be used to express the internal structure of a SC, so that the interaction among the internal tasks of the component is carried out following a static and predictable pattern.

- *multiblock* is a pattern used for the solution of domain decomposition and multi-block problems.
- *farm* improves the throughput of a task, as the different data sets can be computed in parallel.
- *pipe* is used to pipeline a sequence of tasks that communicate using array interchange.

In addition, components with a structure that does not fit these skeletons are also considered. In this case, they are dealt with as (sequential or data-parallel) black boxes where the only information available is related to input/output arguments.

The code below shows an example of the interface of an hypothetical component named `my_sc`.

```
CONFIGURATION INTERFACE my_sc
  STREAM, complex, INOUT, DOMAIN2D :: a
  DISTRIBUTE IN a(*,BLOCK)
  DISTRIBUTE OUT a(BLOCK,BLOCK)
  STRUCTURE
    PIPE my_pipe
      c1(a) ON PROCS(2)
      FARM my_farm (?n) c2(a) ON PROCS(1)
      c3(a) ON PROCS(4)
    END
  END
END
```

The SC accepts an input stream of matrices (of base type `complex`) and produces an output stream of the same type. Input data are distributed by columns and output data by regular blocks. The interface also expresses that the SC is internally structured as a *pipe* of other SCs. The second stage of this pipeline is replicated by means of a *farm* structure. This can be very useful in the case where `c2` is a sequential SC. The parameter `n` indicates that the total number of replicas can be established at composition time. For the other SCs, the number of processors is indicated. A graphical description of `my_sc` is shown in Fig. 1, where the component is being executed using nine processors in total (in this case, the number of replicas of the *farm* equals three).

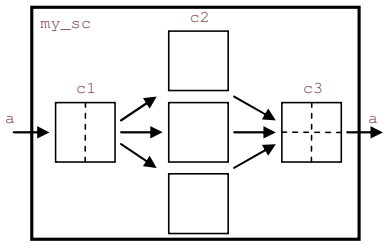

Figure 1. Pipe- and farm-based structure

The component composition language syntax is very similar to the one used in SC interfaces. In other words, SBASCO applications are set up as groups of SCs executed on disjoint sets of processors and coordinated by the skeleton definitions.

2.2. Aspects, Join Points and Connectors

The adding of aspects is carried out by defining an additional layer on top of the existing elements of SBASCO, which do not change their original meaning.

Up to this point, SCs are the only way to design the tasks of a numerical problem. These components are coarse grain as they implement complete tasks with possibly a high degree of functionality. Due to this fact, it is likely that a SC suffers from code tangling problems, as different characteristics may be mixed in the same blocks of code. In addition, such characteristics may affect several SCs. By using aspects, these concerns will be programmed outside the SCs. This decreases the complexity of the SCs which become easier to develop and reuse.

A new element called Aspect Component (AC) is defined for encapsulating aspects. An AC offers operations with the aspect implementation (i.e. advice code). These operations will be called at different points of the application control flow. Unlike SCs which use data-flow primitives for the interactions, the ACs are based on traditional method calls.

The idea of using components to model aspects aims to achieve a symbiosis similar to the one that is discussed in proposals dealing with non-parallel models [9][17]. In our context, the SCs do not call the methods of the ACs in an explicit way. Instead invocations are controlled by a new type of element called Aspect Connectors (ACNs) that allow the interaction information to be declared in a separate layer. The SCs themselves may even not be aware of the execution of aspects.

Computational tasks and aspects can be reused more easily due to the component-based approach which enables the pluggability of aspects into applications to add specific modularized concerns.

In order to implement complex characteristics, the ACs will often need to access some properties that belong to the internal state of the SCs. For this reason, the SCs can also supply interfaces to the ACs in such a way that the latter are able to access data of the former. Unlike classical approaches such as AspectJ-like languages, our ACs can not alter the SCs arbitrarily. Instead, the ACs are limited to using only the operations exported by the SCs. This means that SCs can control the effects of executing (non-invasive) aspects.

In summary, apart from the data-flow interface of the SCs, both SCs and ACs now implement the so-called aspect interface that enables both types of component to interact.

We have adopted an IDL language for the aspect interface definition. The IDL is a subset of OMG IDL (used, for example, in CCM). Some of the main elements of CCM (e.g. attributes, events, exceptions) have been excluded since they may be difficult to manage in a high-performance scenario. The data types chosen for the method parameters can be either a predefined type or a reference to an object which will have to be implemented.

Fig. 2 depicts a hypothetical composition of SCs and ACs. The SCs are executed following the pipe algorithmic pattern. In addition, three aspects add functionality affecting different components in the application. For example, at specific points of the control flow either `SC1` or `SC2` can call functions on the AC `A1`. The advice code encapsulated in `A1` may require access to properties of the two SCs, and so the aspect component is allowed to invoke methods on the SCs. Another concern is encapsulated in `A2` and affects only `SC1`. In the figure, the interface operations painted black implement the aspect code. On the other hand, interfaces painted white define the methods offered to the aspects.

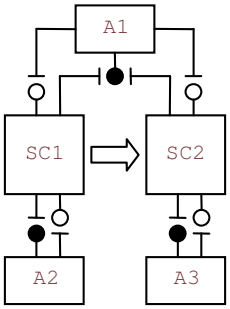

Figure 2. Connecting SCs and ACs

In aspect-oriented languages a join point model establishes the set of points on which it is possible to execute aspect code. A way to define subsets of these join points is typically needed to indicate the cross-cutting nature of the aspects modeled.

Most of the proposals unifying components and aspects use the interface operations as the potential join points. However, in SBASCO the interaction between SCs is carried out only by calling the two data-flow functions `get_data()` and `put_data()`. In order to provide a richer set of valid join points we have identified a collection of fixed points (i.e. internal actions) that are generic for our skeletal applications. These are the only points at which aspect code can be run. It can be an advantage to use join points that are common to a family of applications since the ACs developed become easier to reuse.

Examples of valid join points in our model are: *init*, for data initialization; *create*, for the creation of new instances; *getdata*, for receiving data from other SCs; *putdata*, for sending data to other SCs; *iterate*, for processing the SC main body; *step*, for computing a time step in a multiblock; *converge*, for evaluating local convergence in a multiblock; *resize*, for varying the number of processors used by a SC; *terminate*, for doing resource liberation.

The way to exploit the join points is by means of aspect connectors. This element is used to declare the interactions between SCs and ACs. The advantages of expressing interaction in a separate unit are improved modularity and better AC reuse. The code below shows the scheme of an ACN.

```
ACN acn_name ON COMPONENT list_of_SCs {
  ADVICE [BEFORE|AFTER] ON list_of_actions {
    advice body (AC method invocations)
  }
}
```

An ACN declaration contains a name and a list of SCs to be affected by the aspect. The body of the ACN has advice declarations. An advice indicates an interaction (involving a set of ACs) which will happen before or after the execution control flow reaches any of the points declared in the header. The advice body consists of one or several methods calls to be done on the ACs. The language to express such invocations is C++ although they could be based on any other syntax.

To summarize, the effect of ACNs is to carry out calls to the AC methods at specific points of the SCs. Typically, in every invocation a reference to the caller component is passed so that the AC can, in responde, call the methods on the participant SCs.

3. System Implementation

This section describes the implementation of the aspect logic. First, the original SBASCO implementation is described. Then, the mechanisms are extended to support the efficient execution of aspects. Although this work assumes a distributed memory parallel systems as target platform, the abstractions defined can be implemented on other types of architecture.

3.1. The Original SBASCO System

SBASCO offers the programmer a framework of C++ classes for application development. The most important element of the framework is the class `ScRoot`, which is the base class of every SC. The development of a SC involves the creation of a new subclass of `ScRoot`. The programming of the component consists of coding (overwriting) a collection of virtual methods with the specific scientific code. If a skeleton was used to establish the internal parallel structure, the developer does not need to write parallel code as the SC will be automatically decomposed into the various base SCs. Each one of these SCs can be either, sequential or parallel. A parallel SC which is not based on a skeleton has to be programmed using MPI. `ScRoot` offers operations to access the arguments declared in the component interface.

Another important class is `Framework`. There is one instance of this class (per process) that can be retrieved by any component in the application. `Framework` has methods that return an MPI intra-communicator representing the communication context for a single SC. It should be noted that one SC communicates with others by means of `get_data()` and `put_data()` primitives. These functions act hiding the complexity of parallel communication which is managed by the runtime system. A SC typically has to receive input data by calling `get_data()`, then perform a computation, and finally set the values of the output arguments, which are sent to other SCs via `put_data()`. The `Framework` instance provides a data of class `CommScheme` that encapsulates all the parameters that describe the application structure: the SCs being used, the number of processors assigned to each component, the distribution of data, the skeletons applied, and so on. This type of information is useful to determine, at runtime, the SCs that participate in a communication as well as the data that should be sent to other peer components.

A SBASCO application is executed as a set of MPI processes. Specifically, one MPI application is used to implement each one of the SCs. If a SC is a parallel component its corresponding MPI application will be executed using `N` processes, the parameter `N` being indicated by using the composition language.

An additional application, called *manager*, is in charge of starting the components on disjoint sets of processors. Applications are launched using `MPI_Comm_spawn()`. Manager interacts with the SCs to send them the `CommScheme` object. As soon as the SCs receive this information, they can establish the corresponding connections using `MPI_Comm_accept()` and `MPI_Comm_connect()`. The SCs use the information encapsulated in the `CommScheme` instance to achieve efficient point to point data communication.

3.2. Aspect Layer Implementation

The existing implementation mechanisms have been extended to consider the execution of aspect components preserving, at the same time, the performance of applications.

Our ACs are implemented using standard C++ classes. The class that represents an AC has to provide aspect code for the different method interfaces declared in IDL. Furthermore, the SCs (or more specifically, the subclasses of `ScRoot`) have to implement the methods (declared in IDL) that enable aspects accessing the SC internal state.

The next step is to implement an efficient weaving mechanism by which aspects can be mixed with the base code in order to compose final applications. A new group of virtual functions is added to the class `ScRoot`. To be precise, a new pair of functions is added for each one of the predefined join points. As an example, the point *getdata* is the origin of the two virtual functions: `before_getdata()` and `after_getdata()`. These methods are initially "empty". They have to be overwritten in the subclasses of `ScRoot` with code for specific aspect interactions. In AOP, the tool in charge of combining aspects and base code is called *aspect weaver*. In our system, the weaver is a source-to-source compiler that proceeds as follows: for each aspect connector being declared, the source code of the participant SCs (the ones being affected) is changed in such a way

that the advice body (method calls on the ACs) is inserted into the corresponding methods. For instance, if an ACN on a component SC1 declares one advice of type *before* on the point *getdata*, the interaction code of the advice will be copied into before_getdata() in the class that implements SC1. During execution, each time the user invokes get_data(), the code of before_getdata() will be triggered first. As a result, the method call on the AC is performed.

In Section 2.2, Fig. 2 showed a composition example. Such a composition has the run-time structure in terms of processes and class instances depicted in Fig. 3.

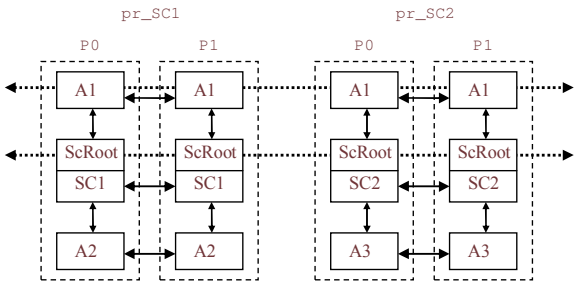

Figure 3. Structure of processes

Fig. 3 clarifies many of the details about the execution of SBASCO applications and the new aspect layer. There are two MPI programs in total, called pr_SC1 and pr_SC2. Each one is associated to one single SC. We are assuming that each SC is being executed using two processes (P0 and P1). Inside a process, one single instance of the SC is managed. In addition, the set of ACs that affect such a SC are also instantiated. For example, if A1 can invoke a method on SC1, an instance of the former will be created in every process that contains the latter. This structure is replicated, as observed in the figure. Since the aspect A2 can only call methods on SC1, the former is created in the processes running pr_SC1 only.

In addition, Fig. 3 also depicts the different types of communication which can happen in our applications. Communication is represented by arrows in the figure. Let us review each scenario. Firstly, vertical arrows indicate that the SCs can invoke operations on the ACs. As described before, this type of interaction is always triggered by the ACN declarations. If an advice causes the SC to call a method on an AC, the call will be done, in parallel, in all processes that contain the two entities. As can be observed, the overhead of executing aspects is only the invocation of an additional C++ virtual function, followed by a standard (non-virtual) call. This is a major advantage of our execution environment. We are not applying any type of complex runtime structure to support aspects. Weaving is done statically, and the cost of invoking aspect functionality can be considered negligible, as is to be expected in high-performance computing scenarios.

Horizontal arrows in Fig. 3 are inter-process communications based on message passing. Both, SCs and ACs, can exploit internal parallelism. For that purpose, they can implement the parallelism using a MPI intra-communicator obtained from the Framework instance. This context is represented by the short solid arrows. The longer dotted arrows represent interactions between different sets of processes based on inter-communicators. On the one hand, the SCs use this interaction by calling get_data() and put_data(). In this case, the communication is automatically managed by the system in accordance with the application structure and the skeletons used. On the other hand, the ACs may also exploit this type of interaction. To do that, the interface of Framework is extended so that not only intra-, but also inter-communicators can be retrieved. This extension is not difficult to implement because the system already uses these objects internally for the data-flow communication. Once aspects can access the structure of communicators, they are able to carry out data communication between different groups of processes (obviously, an AC only can communicate the groups of processes in which it is instantiated).

The mechanisms described allow aspects to carry out a wide variety of interactions. For example, let us focus on aspect A1 of Fig. 3. This component may perform a simple local computation and access the SC to modify some values without the need for communication. The AC may also implement a parallel computation inside each group of processes. Finally, the aspect may implement some sort of complex parallel interaction, retrieving data from processes running pr_SC1, then sending values to processes running pr_SC2, and finally accessing the interface of SC2 to fill some properties using the values received from the other group of processes.

4. Application Example: 2D-FFT

This section shows a simple example that uses many of the elements defined. The application proposed computes the two-dimensional Fast Fourier Transform (2D-FFT) [3] in parallel. This is a kernel widely used in multiple domains such as image and signal processing. The 2D-FFT of a matrix can be solved using (one-dimensional) FFT on the columns of the matrix first, and then using FFT on the rows of the result. A pipeline structure composed by two data-parallel stages represents an efficient and scalable solution.

The goal is to compute 2D-FFT of a set (input stream) of matrices which can be read from I/O (e.g. database, file, CORBA server). Our purpose is the separation of the I/O concern from the numerical code. Two data-parallel SCs, named cfft and rfft, are used to compute FFT on columns and rows, respectively. An AC, named dataIO, manages the way in which the input data are acquired and the results saved. The resultant design has a high degree

of modularity in the three dimensions: all the numerical code is written in the SCs; the interaction between the two data-parallel tasks is expressed by the pipe pattern; the characteristic of I/O, which affect both SCs, is implemented in the AC. By replacing the AC, we can plug different I/O mechanisms into the application.

The following code describes the configuration interface of the SCs.

```
CONFIGURATION INTERFACE cfft
  STREAM, complex, OUT, DOMAIN2D :: a
  DISTRIBUTE OUT a(*,BLOCK)
END

CONFIGURATION INTERFACE rfft
  STREAM, complex, IN, DOMAIN2D :: a
  DISTRIBUTE IN a(BLOCK,*)
END
```

The solution based on the SBASCO composition language is designed as follows.

```
PROGRAM 2D-FFT
  integer :: nrow = 512, ncol = 512
  STREAM,complex,DOMAIN2D::a/1,1,nrow,ncol/
  STRUCTURE
    PIPE my_2dfft_pipe
      cfft(a) ON PROCS(4)
      rfft(a) ON PROCS(4)
    END
  END
END
```

The code above indicates the size of the matrix and the number of processors to execute each one of the SCs. Up to this point, no aspects have been considered yet.

The next step is the declaration of `dataIO` and the aspect interfaces of both the AC and the SCs. The following code is expressed using the IDL.

```
interface IOState {
  void initGet(in Sc sc);
  void initPut(in Sc sc);
  void initCommScheme(in Sc sc);
};

interface IOFunctions {
  void setInput(in Domain2D m);
  Domain2D getResult();
};

interface Configuration {
  void go(in CommScheme cs,in unsigned mts);
  void setDataIO(in dataIO dio);
  dataIO getDataIO();
};

component dataIO {
  provides IOState state;
  uses IOFunctions funcs;
};

component Sc {
  provides IOFunctions funcs;
  provides Configuration con;
  uses IOState state;
};

component cfft : Sc {};
component rfft : Sc {};
```

The AC `dataIO` implements `IOState`, which is the interface that declares the aspect operations. The type of component `Sc` is the base class for `cfft` and `rfft`. This enables the definition of functions which are common to both SCs. The SCs implement the `IOFunctions` and `Configuration` interfaces. The first one has methods to get a result and to set new input data, while the second one has methods needed to start the computation and to set an instance of type `dataIO`.

The next step is the definition of the connectors (ACNs). The code below declares the interaction with the aspect component.

```
ACN scheme_acn ON COMPONENT Sc {
  ADVICE BEFORE ON init_call {
    getDataIO()->initCommScheme(this);
  };
};

ACN input_acn ON COMPONENT cfft {
  ADVICE BEFORE ON iterate_call {
    getDataIO()->initGet(this);
  };
};

ACN output_acn ON COMPONENT rfft {
  ADVICE AFTER ON iterate_call {
    getDataIO()->initPut(this);
  };
};
```

The advice code declared in the ACNs is statically woven into the SCs. For example, `scheme_acn` causes a call to `initCommScheme()` on `dataIO` when the application starts. The implementation of this method accesses the `CommScheme` object to set up the AC. The ACN `input_acn` indicates that every time `cfft` processes its main body, a call to `initGet()` on the AC is performed. In response, `dataIO` reads a new matrix from I/O, then divides the data according to the information of the object `CommScheme` (in the case where data are read only from one process, `dataIO` internally communicates the data pieces to the rest of the processes). The final step is to invoke `setInput()` on `cfft`. To save the result, the ACN `output_acn` triggers the execution of `initPut()`, which collects and stores distributed output data from the processes that host `rfft`.

5. Conclusions

This paper describes a way to extend the high-performance component model of SBASCO. The objective is to define new concepts and abstractions for the management of cross-cutting functionality using aspect-oriented programming. The resulting model uses a type of component, the so-called aspect component, to implement the application concerns in a modular way. The aspect components interact with the original scientific components by means of aspect connectors, which exploit a simple join point model to express the cross-cutting nature of aspects. The ACs can be either sequential or parallel, and they can implement different types of communication styles. By using components to manage both computational tasks and aspects, applications are easier to develop and reuse.

Besides the abstractions introduced, an efficient implementation of the high-level mechanisms based on MPI and focussed on distributed memory parallel systems is presented. Our aspects interact with the base code efficiently since the overhead of aspect execution is equivalent to the cost of a C++ virtual function call, followed by a standard C++ method call on the corresponding object that implements the AC. Both invocations are executed locally. The aspect developer is free to implement parallelism in the ACs if required.

References

[1] R. Armstrong, G. Kumfert, *et. al.*, The CCA Component Model for High Performance Scientific Computing, *Concurrency and Computation: Practice and Experience*, **18** (2), pp. 215 - 229, 2006.

[2] A. Benoit, M. Cole, S. Gilmore, J. Hillston, Flexible Skeletal Programming with eSkel, in Proc. of the 11*th* International Euro-Par Conference, Lisboa, Portugal, LNCS 3648, pp. 761 - 770, 2005.

[3] E.O. Briham, *The Fast Fourier Transform and Its Applications*. Prentice-Hall International, 1988.

[4] M. Danelutto, QoS in Parallel Programming Through Application Managers, in Proc. of the 13*th* Euromicro Conference on Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, pp. 282 - 289, 2005.

[5] M. Díaz, B. Rubio, E. Soler, J.M. Troya, SBASCO: Skeleton-Based Scientific Components, in Proc. of the 12*th* Euromicro Conference on Parallel, Distributed and Network-Based Processing, A Coruña, Spain, pp. 318 - 324, 2004.

[6] M. Díaz, S. Romero, B. Rubio, E. Soler, J.M. Troya, Using SBASCO to Solve Reaction-Diffusion Equations in Two-Dimensional Irregular Domains, in Proc. of the 3*rd* International Workshop on Practical Aspects of Parallel Programming, Reading, UK, LNCS 3992, pp. 912 - 919, 2006.

[7] A. Dorta, P. López, F. Sande, Basic Skeletons in LLC, *Parallel Computing*, **32** (7-8), pp. 491 - 506, 2006.

[8] T. Elrad, R. Filman, A. Bader, Aspect-Oriented Programming: Introduction, *Communications of the ACM*, textbf44 (10), pp. 29 - 32, 2001.

[9] M. Frantz, A. Gal, D. Beuche, Learning From Components: Fitting AOP for System Software, in Proc. of the 2*nd* AOSD 2003 Workshop on Aspect, Components and Patterns for Infrastructure Software, Boston, USA, pp. 51 - 55, 2003.

[10] G. Heinemann, W. Council, *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[11] B. Harbulot, J. Gurd, Using AspectJ to Separate Concerns in Parallel Scientific Java Code, in Proc. of the 3*rd* International Conference on Aspect-Oriented Software Development, Lancaster, UK, pp. 122, 2004.

[12] B. Harbulot, J. Gurd, A Join Point for Loops in AspectJ, in Proc. of the 5*th* International Conference on Aspect-Oriented Software Development, Bonn, Germany, pp. 63 - 74, 2006.

[13] G. Kiczales, E. Hilsdale, *et. al.*, An Overview of AspectJ, in Proc. of the Europe Conference on Object-Oriented Programming, Budapest, Hungary, LNCS 2072, pp. 327 - 353, 2001.

[14] S. Pelagatti, *Structured Development of Parallel Programs*. Taylor and Francis, 1998.

[15] C. Pérez, T. Priol, A. Ribes, PaCO++: A Parallel Object Model for High-Performance Distributed Systems, in Proc. of the 37*th* Hawaii International Conference on System Sciences, Hawaii, USA, pp. 274, 2004.

[16] J.L. Sobral, Incrementally Developing Parallel Applications with AspectJ, in Proc. of the 20*th* International Parallel and Distributed Processing Symposium, Rodhes, Greece, pp. 10, 2006.

[17] D. Suvée, B. Fraine, W. Vanderperren, A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development, in Proc. of the 9*th* International SIGSOFT Symposium on Component-based Software Engineering, Stockholm, Sweeden, LNCS 4063, pp. 114 - 122, 2006.

[18] M. Vanneschi, The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications, *Parallel Computing*, **28** (12), pp. 1709 - 1732, 2002.