An Aspect Oriented Framework for Scientific Component Development*

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

Abstract

Aspect-Oriented Programming enables developers to capture in separated aspect modules concerns that are spread over different components in a system. This paper is an attempt to apply this paradigm to High Performance Computing. Besides achieving the usual advantages of improved modularity, more reusable code that is easier to develop and maintain, we pursue to improve efficiency by means of dynamic changes of aspects at runtime. We present an Aspect-Oriented Framework where Scientific Components and Aspects are first-order entities (components) which interaction is established by means of what we have called Aspect Connectors. As an example, we focus on the communication aspect, which encapsulates the communication scheme among the set of components that set up a system. The expressiveness and suitability of the approach are shown by means of an application example.

1. Introduction

Component-Oriented Programming, initially applied to the business world, is coming to the high-performance parallel and distributed computing scene. Component standards and implementations, such as OMG CCM [11], Microsoft DCOM [6], Sun Java Beans and Enterprise Java Beans [5] [10], share serious shortcomings for parallel and distributed scientific applications, due to the lack of the abstraction needed by parallel and distributed programming and poor performance. They also have trouble with the mechanism for encapsulating an existing scientific application (which might itself be a parallel-distributed application) into a component.

* This work was supported by the Spanish project MICYT TIC 2002-04309-C02-02

Recently, some efforts are being carried out in order to incorporate component technologies to develop parallel and distributed Programming Environments (PEs) in the high-performance computing area. In this sense, ASSIST [17] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. SBASCO [4] is a PE oriented to the efficient development of parallel and distributed numerical applications. A large effort is currently devoted to define a standard component architecture for high-performance computing in the context of the Common Component Architecture (CCA) Forum [16].

This component-oriented approach results in more reusable, extensible and adaptable software. However, achieving these features requires an appropriate separation of the system concerns in independent modules. When designing and building systems, it is often difficult to achieve a design that modularises all required system features. Commonly, there are certain concerns that will not fit well into any component structure we choose. This is particularly true for certain kinds of functionality, such as logging, debugging, communication, synchronization, distribution, security, etc. Design compromises often lead to code where the same concern happens to be spread over different components in a system. These concerns are usually called crosscutting concerns.

Aspect-Oriented Programming (AOP) [7] is a paradigm that enables developers to capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has a greater potential for reuse. A well-modularised crosscutting concern is called an aspect.

Current AOP technologies offer different alternatives to tackle the separation of concerns, that differ mainly in three factors: the aspect definition language, the weaving process, and the kind of concerns modeled as aspects. AO languages can be aspect specific [9] or extensions of general purpose

languages providing special constructions used to implement any kind of aspects [8]. One common drawback of AO languages is that the weaving process is static, mixing component and aspect code at compile-time. One alternative to AO languages are AO frameworks [13]. Usually, AO frameworks model components and aspects as separated entities which are implemented in the same general purpose language and composed in a more o less dynamical way at runtime.

This paper presents an Aspect-Oriented Framework as an attempt to apply the AOP paradigm to the high-performance computing area. Our application domain is the parallel and distributed solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains, and other kinds of problems that take advantage of integrating task and data parallelism and have a communication pattern based on (sub)array interchange. Usually, their execution times are large.

In this framework there are two kinds of component: Scientific Components (SCs), which are in charge of computational tasks, and Aspect Components (ACs) used to encapsulate crosscutting concerns. This way, we pursue two benefits. On the one hand, those usual of AOP, simpler and more reusable code that is easier to develop and maintain. On the other hand, to improve application execution time by means of establishing changes at the aspect level (a better communication scheme, distribution, etc) dynamically at runtime.

SCs interact each other following a ”data flow” style by means of a typical put_data/get_data scheme based on the SBASCO approach. The interaction among SCs and ACs follows a ”procedural” style and is governed by means of what we have called Aspect Connectors (ACNs). The code of an ACN is statically (at compile-time) woven to a SC, while, as stated above, an AC can dynamically change its characteristics. In addition, a different AC can be composed at runtime.

Our approach provides the basis to separate specific aspects that appear in the application domain, such as communication, synchronization, distribution, convergence criteria, etc. As an example, we focus in this paper on the communication aspect, which encapsulates the communication scheme among the set of SCs that build a system.

The rest of the paper is structured as follows. Next section outlines the main characteristics of SBASCO taken into consideration in our approach. Section 3 explains the SCs implementing the computational task of a system and the ACs encapsulating the communication aspect among SCs. The way the ACNs are used to connect SCs and ACs is shown in section 4. Finally, some conclusions and future work are sketched in section 5.

2. An Overview of SBASCO

The inter-connection among SCs considered in our framework is based on SBASCO, a programming environment oriented to the efficient development of parallel and distributed numerical applications. This environment integrates two technologies: skeletons and components. A software skeleton is a known reusable parallelism exploitation pattern [1]. A structured parallel programming methodology is achieved by means of the establishment of a fixed set of software skeletons or parallelism constructors, which are the only way to express the parallel structure of the programs [12].

In SBASCO, the internal structure of a component can be established by means of the utilization of the three different skeletons previously defined in DIP [3]:

- The *multi-block* skeleton is focused on the solution of multi-block and domain decomposition-based problems, which constitute an important class of problems in the high-performance computing area.
- The *farm* skeleton improves a task throughput as different data sets can be computed in parallel on different sets of processors.
- Problem solutions that have a communication pattern based on array interchange and may take advantage of integrating task and data parallelism, can be defined and solved in an easy and clear way by using the *pipeline* skeleton, which pipelines sequences of tasks in a primitive way.

A skeleton definition is based on the concept of *domain* [2]. Basically, a domain consists of the Cartesian points establishing a region.

The following code shows the general scheme of the `MULTIBLOCK` skeleton. Tasks solve the different domains the problem has been decomposed in (Figure 1).

```
MULTIBLOCK skeleton_name
 task1(arguments) processor layout
 task2(arguments) processor layout
 .....
 taskm(arguments) processor layout
WITH BORDERS
 border definitions
END
```

In the `FARM` skeleton, the stream of input data for the task is accepted by a special process called emitter which schedules them to each replica. On the other hand, a collector process collects the results from each replica and merges them into the stream of output data from the task (Figure 2).

This skeleton is specified as follows:

```
FARM skeleton_name (R) task(arguments)
     processor layout
```

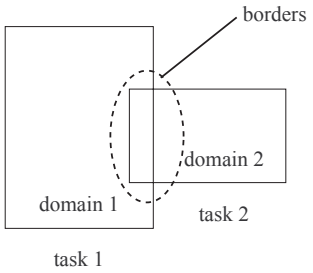

Figure 1. A problem decomposed in two domains

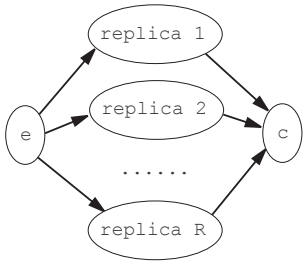

Figure 2. Structure of a farm with R replicas

where `R` establishes the number of replicas and `processor layout` indicates the number of processors where each replica is going to be executed.

The `PIPE` skeleton pipelines sequences of tasks. Figure 3 depicts the structure of the general n-stage pipeline corresponding to the `PIPE` skeleton shown in the following code:

```
PIPE skeleton_name
  stage1
  stage2
  .....
  stagen
END
```

Each stage in the pipeline consumes and produces a data stream, except the first and the last stages that may only produce and consume, respectively. The communication between two pipe stages is specified by using the same domain as argument in both stages, unlike in the `MULTIBLOCK` skeleton, where border definitions are used.

In SBASCO, there are two views of a component interface. The *application view* contains information related to data types of component input/output. This view is used by the programmer in order to develop his/her applications by means of a composition language. The *configuration view*

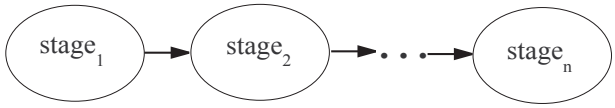

Figure 3. Structure of the n-stage pipeline

extends the application view with information about input and output data distribution, processor layout and component internal structure (in terms of skeleton composition scheme).

A configuration tool uses the latter view, together with an skeleton-associated cost model, to obtain an efficient implementation of the application on parallel/distributed platforms. The knowledge at the component interface level of data distribution and processor layout allows the system to obtain an efficient implementation of the communication scheme among components, which follows a ”data flow” style by means of a typical put_data/get_data scheme. A more detailed explanation of the two different views and the way they are used can be found in [4].

3. Scientific and Communication Aspect Components

The previously mentioned SBASCO configuration tool tries to establish the most appropriate system configuration. In order to achieve this, the analytical cost model associated to each software skeleton is enhanced by a runtime analysis focused on the establishment of some parameters, such as the number of processors used for each task and the number of replicas in a farm, which are of capital importance to obtain an efficient system implementation.

Due to the fast evolution of hardware of both processors and inter-processor communication systems, an accurate estimation is quite difficult. In addition, the system evolution during execution may require a readjustment of that kind of parameters in order to maintain the efficiency. This way, the communication scheme among the different components, which depends on this information, should be modified. But in the SBASCO system, the communication scheme is a crosscutting concern, that is, it is spread over the components.

The main goal of our framework is to improve efficiency by means of establishing changes on this kind of concerns dynamically at runtime. In order to achieve that, we capture them in separated (aspect) components. Besides efficiency, we obtain more reusable code that is easier to develop and maintain.

In our approach, Communication Aspect Components (CACs) encapsulate the communication scheme among a

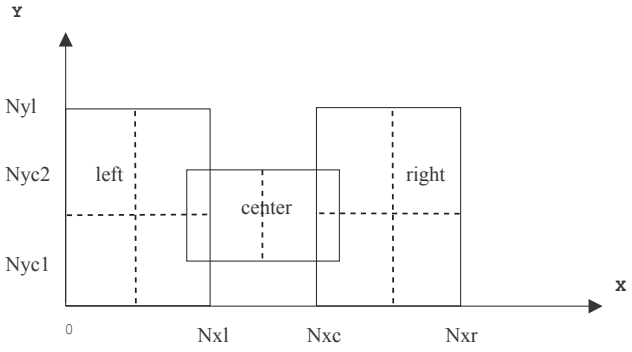

Figure 4. A domain decomposition of an irregular geometry

set of SCs of a system. This scheme is captured by the corresponding skeleton, where data distribution, processor layout and, possibly, number of replicas of each SC are specified.

In order to show the way both kinds of components are used, we introduce an example that solves a system of two non-linear reaction-diffusion equations in an irregular geometry. A detailed explanation for this problem and the employed numerical method can be found in [14].

Figure 4 depicts the irregular geometry of the problem that has been decomposed in three domains (`left`, `center`, `right`). Each domain is solved by a SC. Figure 5 shows their definitions together with the description of the CAC that captures their communication scheme. We use a syntax based on CCM [11], enriched by constructors and data types characterizing the programming model of SBASCO.

The definition of the CAC, besides specifying the provided and used interfaces, establishes the `Multiblock` skeleton that describes the interaction scheme among the SCs. The predefined type `Domain2D` is used to capture the Cartesian points that establish the region of a domain. For example the expression

```
Domain2D left = {0,0,Nxl,Nyl};
```

declares a two-dimensional domain `left` for the region of the plane that extends from the point `(0,0)` to the point `(Nxl,Nyl)`.

On the other hand, the type `CommScheme` will contain the required information, which is declared in the skeleton, in order to carry out an efficient inter-component communication: data distribution, interaction among domains and mapping of processors.

The data distribution types are HPF-like. For example, the expression `center:(*,BLOCK)` declares a distribution by columns for the domain `center`. This declaration indicates the data distribution inside the SC that is bound to the specified domain. In Figure 4, dotted lines represent data distributions.

```
interface ScConfigure {
    void go(in Domain2D d,
            in CommScheme scheme, in unsigned numProcs);
    void setCacGet(in Cac cacGet);
    void setCacPut(in Cac cacPut);
};
interface ScChangeConf {
    void changeCommScheme(in CommScheme scheme);
    void changeNumProcs(in unsigned numProcs);
};
interface CacProvided {
    void initGet(in Sc sc);
    void finishGet(in Sc sc);
};

component Sc {
   provides ScConfigure conf;
   provides ScChangeConf change;
   uses CacProvided setStateInf;
};

component SolveL : Sc {};
component SolveC : Sc {};
component SolveR : Sc {};

component Cac {
   skeleton ReactionDiffusion {
        Domain2D left = {0,0,Nxl,Nyl},
                 center = {Nxl-1,Nyc1,Nxc+1,Nyc2},
                 right = {Nxc,0,Nxr,Nyl};
        Multiblock {
            SolveL(left:(BLOCK,BLOCK)) ON PROCS(2,2);
            SolveC(center:(*,BLOCK)) ON PROCS(2);
            SolveR(right:(BLOCK,BLOCK)) ON PROCS(2,2);
        WithBorders
            left(Nxl,Nyc1 ,Nxl,Nyc2) <- center(_);
            center(Nxl-1,Nyc1 ,Nxl-1,Nyc2) <- left(_);
            center(Nxc+1,Nyc1, Nxc+1,Nyc2) <- right(_);
            right(Nxc,Nyc1, Nxc,Nyc2) <- center(_);
        };

   };
   provides CacProvided setStateInf;
   uses ScChangeConf change;
};
```

Figure 5. Definition of SCs and CAC for the reaction-diffusion example

Another information related to domains is the interaction information among domains (borders). In general, the problem is solved iteratively, so that for each iteration a communication of the different borders is needed. This communication is carried out by means of the predefined data flow primitives `get_data` and `put_data`, which obtain data from and send data to a SC, respectively. Borders are defined among the specified domains. For example, the expression

```
left(Nxl,Nyc1,Nxl,Nyc2) <-
center(Nxl,Nyc1,Nxl,Nyc2);
```

indicates that the zone of `left` delimited by points `(Nxl,Nyc1)` and `(Nxl,Nyc2)` will be updated by

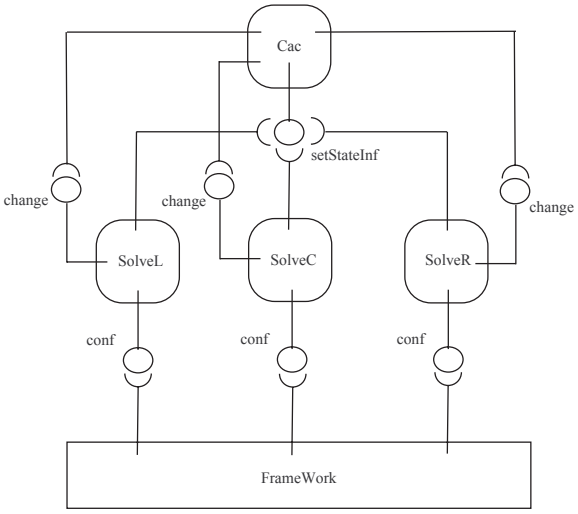

Figure 6. Interaction scheme for the reaction-diffusion example

the values belonging to the zone of `center` delimited by the same points. In our example, the domains have been defined taking into account their distribution in the plane. Thus, in the definition of the borders, the regions at both sides of the operator `<-` are the same. In these cases, our approach provides the notation `_` to be used at the right hand side.

In addition to data distribution and borders, the relationship between domains and the processor layout where the SCs are going to be executed are also indicated. Each SC is solved by a disjoint set of processors. For example, the expression:

```
... ON PROCS(2,2);
```

indicates a square arrangement of 4 processors.

Figure 6 depicts the interaction scheme among the different components. The sequence of application execution begins with the creation of the three SCs and the corresponding CAC. The framework calls the methods `setCacGet` and `setCacPut` for each SC, so that these obtain a reference to the CAC for further communication. Our approach provides two different methods to set a CAC for a given SC because certain applications may require distinct CACs for both `get_data` and `put_data` data flow primitives. In this example, however, the communication scheme is the same for both `get_data` and `put_data` primitives, and so, the provided CAC is also the same. Then, the `go` method is invoked, so that the domain, the initial communication scheme and the number of processors of each SC are established. After that, each SC begins its computational task.

During the application execution it is possible that the CAC decides to change some parameters that affect the communication scheme, such as the number of processors of a SC. In this case, the CAC will call the `changeNumProcs` method for the target SC, and the `changeCommScheme` method for the SCs which communication scheme is affected by this change. For example, if the number of processors of `SolveL` changes, this will imply a change in the communication scheme for `SolveL` and `SolveC`, but not for `SolveR` (see Figure 4). In our application domain, where we usually face to problems with large execution times, the time spent on this kind of changes is clearly recovered by the efficiency obtained applying them. On the other hand, a different CAC could be composed at runtime. This way, the methods `setCacGet` and `setCacPut` would be invoked again.

The way a CAC obtains the necessary information in order to decide whether a parameter is changed or not is explained in next section.

4. Interconnecting SCs and CACs

In AOP it is needed a tool called aspect weaver in order to insert code fragments (advice) that are derived from the aspect code to wherever they are needed. These insertion points are called join points and must be established by the programmer. A join point refers to a method, an attribute, a class, an object, etc. The join points, and the way they are grouped, are the key language elements to deal with the crosscutting nature of aspects. They can describe points in the static structure or the dynamic control flow of a program in a very flexible way. The joint points and the advice that should be run when they are reached constitute the way an aspect is associated to a code.

General purpose AO languages, such as AspectJ [8] and AspectC++ [15], are expressive enough to implement any kind of aspect. They offer expressions and algebraic operators in order to map, filter and combine different joint points. Moreover, different kinds of advice can be declared, including ”after” advice that runs after the joint point, ”before” advice that is executed before the join point and ”around” advice, which is executed in place of the join point.

In our framework, components and aspects are separated entities that are associated by using the methods `setCacGet` and `setCacPut` in the way described in previous section. However, after association, SCs and CACs must interact each other during computation. In order to be flexible and expressive enough, our approach allows the programmer to establish this interaction control by means of what we have called Aspect Connectors (ACNs). These ACNs are based on the joint points and advice mechanism of AOP.

The specified code in an advice of an ACN is inserted (woven) to a SC in the target join point. The weaving process is done statically at compile-time.

The special purpose of our proposal makes our scheme less ambitious and simpler than those of general purpose AO languages. We are focused on a reduced set of specific aspects that can appear in our application domain, such as communication, distribution and convergence criteria, and, on the other hand, the joint points we are mainly interested in are interaction points where SCs communicate and synchronize each other following a data flow scheme.

In the example introduced in previous section, the CAC needs to obtain information from SCs in order to establish possible changes in the communication scheme. For example, the CAC could detect that two interacting SCs are running at different speeds, and so, the processors where one of them is executed may be idle for long times. A solution could be to increase the number of processors of the slowest SC. The execution time of a SC iteration can be determined by the elapsed time between the termination of a `get_data` call, carried out in order to get data from the other SC, and the beginning of the following `get_data`. This way, a `get_data` call could be a good candidate to become a join point, where to associate an advice in order to establish a communication between the SC and the corresponding CAC.

Figure 7 shows the ACN to be associated to the `Sc` component specified in Figure 5. The three components (`SolveL`, `SolveC` and `SolveR`) derived from `Sc` inherit it. The ACN establishes two kinds of advice on the same join point, a `get_data` call. The before advice declaration specifies that before calling the `get_data` operation, the SC must inform the corresponding CAC of this situation by means of an `initGet` method call, which argument is a reference to the caller SC. On the other hand, the after advice declaration establishes that after `get_data` operation has been carried out, the SC informs the CAC by invoking the `finishGet` method. This way, the CAC has enough information to take into consideration a possible change concerning the communication scheme among SCs.

5. Conclusions and Future Work

An Aspect-Oriented Framework where components and aspects are separated entities has been presented as an approach to apply the Aspect Oriented Programming paradigm to the high-performance computing area. The interaction among Scientific Components, which are in charge of computational tasks, and Aspect Components, used to capture the crosscutting concerns, is controlled by means of Aspect Connectors, which are based on advice and joint points mechanism. The paper has focused on the communication aspect among SCs. By means of an example, we have shown the expressiveness and suitability of the proposal.

```
ACN on component Sc {
   advice before on get_data_call {
       cacGet.initGet(this);
   };

   advice after on get_data_call {
       cacGet.finishGet(this);
   };
};
```

Figure 7. An ACN for the reaction-diffusion example

We are currently developing a prototype using C++ as the programming language for both Scientific and Aspect Components. The parallel programming of a SC is being carried out by means of the message passing library MPI. Moreover, this library is also used as the runtime support for implementing the `get_data` and `put_data` primitives governing the interaction among SCs.

As future work, we are interested in studying the possibilities of approaching our proposal to a standard for high-performance computing community. In this sense, we think that the Common Component Architecture approach could take advantage of the aspect oriented scheme.

References

[1] Cole, M., "Algorithmic Skeletons: Structured Management of Parallel Computation", MIT Press, Cambridge, MA, 1989.

[2] Díaz, M., Rubio, B., Soler, E., Troya, J.M., A Border-based Coordination Language for Integrating Task and Data Parallelism, *Journal of Parallel and Distributed Computing*, **62**, (2002), 715–740.

[3] Díaz, M., Rubio, B., Soler, E., Troya, J.M., Domain Interaction Patterns to Coordinate HPF Tasks, *Parallel Computing*, **29**, 7 (2003), 925–951.

[4] Díaz, M., Rubio, B., Soler, E., Troya, J.M., SBASCO: Skeleton-Based Scientific Components, in "Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2004)", pp. 318–324, IEEE Computer Society, A Coruña, Spain, 2004.

[5] Englander, R., "Developing Java Beans", O'Really&Associates, 1997.

[6] Horsmann, M., Kirtland, M., "DCOM Architecture" Microsoft White Paper, 1997. Available from http://www.microsoft.com/com/wpaper.

[7] Kiczales, G., et. al., Aspect-Oriented Programming, in "Proceedings of the Europe Conference on Object-Oriented Programming (ECOOP 1997)", LNCS 1241, pp. 220–242, Springer, Jyvskyl, Finland, 1997.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., An Overview of AspectJ, in ”Proceedings of the Europe Conference on Object-Oriented Programming (ECOOP 2001)”, LNCS 2072, pp. 327–353, Springer, Budapest, Hungary, 2001.

[9] Lopes, C.I.V., D: A Language Framework for Distributed Programming, Ph.D. thesis, Northeastern University, Nov. 1997.

[10] Monson-Haefel, R., ”Enterprise Java Beans 3th edition”, O’Really&Associates, 2001.

[11] Object Management Group (OMG), ”Specification of Corba Component Model (CCM)”. http://www.omg.org/technology/documents/formal /components.htm.

[12] Pelagatti, S., ”Structured Development of Parallel Programs”, Taylor&Francis, London, 1997.

[13] Pinto, M., Fuentes, L., Fayad, M.E., Troya, J.M., Separation of Coordination in a Dynamic Aspect Oriented Framework, in ”Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)”, pp. 134–140, ACM Press, Enschede, The Netherlands, 2002.

[14] Ramos, J.I., Soler, E., Domain Decomposition Techniques for Reaction Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners, *Applied Mathematics and Computation*, **118(2-3)** (2001) 189–221.

[15] Spinczyk, O., Gal, A., Schroder-Preikschat, W., AspectC++: An Aspect-Oriented Extension to C++, in ”Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2002)”, pp. 53-60, Sydney, Australia, 2002.

[16] The Common Component Architecture Forum, home page http://www.cca-forum.org.

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