

# A Process for Separation of Crosscutting Grid Concerns

Paulo Henrique M. Maia<sup>1</sup>, Nabor C. Mendonça<sup>2</sup>, Vasco Furtado<sup>1,2</sup>,  
Walfredo Cirne<sup>3</sup>, Katia Saikoski<sup>4</sup>

<sup>1</sup>Núcleo de Aplicação em Tecnologia da Informação

<sup>2</sup>Mestrado em Informática Aplicada  
Universidade de Fortaleza  
Fortaleza, CE, Brazil

phmendes@gmail.com,  
{nabor,vasco}@unifor.br

<sup>3</sup>Laboratório de Sistemas Distribuídos  
Departamento de Sistemas e  
Computação

Univ. Federal de Campina Grande  
Campina Grande, PB, Brazil

walfredo@dsc.ufcg.edu.br

<sup>4</sup>HP Brasil - POA  
TecnoPUC

Porto Alegre, RS, Brazil

katia.saikoski@hp.com

## ABSTRACT

This paper describes how to explicitly separate crosscutting Grid concerns in a parallel Java application. This process, named *GridAspecting*, uses a restricted subset of the Java threads model for application decomposition, and aspect-oriented programming for allowing parallel execution of the application's threads as Grid tasks. As a result of the process, all Grid-related code is encapsulated in aspects, thus improving the application's modularity. In addition, by relying on Java's native concurrency abstractions the process simplifies the Grid programming model and makes it possible to test a Grid application even without the Grid.

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming – distributed programming, parallel programming.

## General Terms

Algorithms, Design, Experimentation, Languages

## Keywords

Separation of Concerns, Aspect-Oriented Programming, Grid Applications

## 1. INTRODUCTION

The opportunity for sharing computational resources has increased with the possibility of integrating independent networked computers as a platform to execute parallel applications. The main advantage of this approach is the flexibility to allocate a vast amount of resources to a parallel application, considering the thousands of computers interconnected by the Internet, and at a lower cost than traditional alternatives, which usually are based on the use of expensive parallel supercomputers [4].

This approach is known as *Grid computing*, which can be

described as the gathering and sharing of distributed resources in an attempt to create a distributed, dynamic, and highly scalable computational environment capable of solving a variety of complex scientific and commercial problems [5]. Although research on Grid computing has started at academic institutions, it has already gained the attention of large computer manufacturers such as HP, Sun and IBM [4], and several Grid platforms are now available for general use (e.g. Globus [7] and OurGrid [2]).

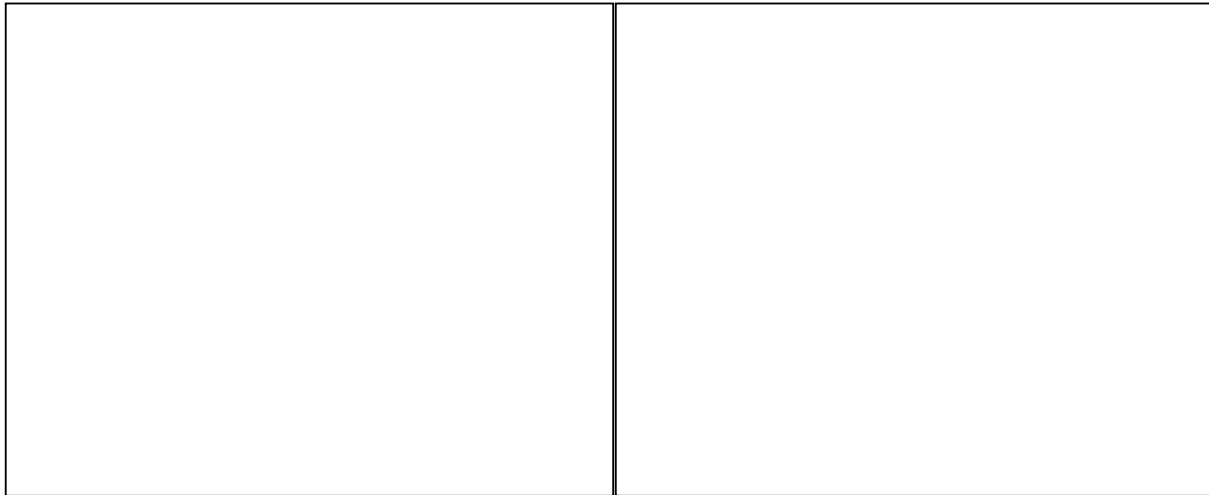
As with any distributed application that relies on specific middleware services, a Grid application must be carefully implemented so as to avoid the dependency of a specific Grid platform. Such dependency is undesirable because it would make it difficult to maintain the Grid application, for instance, to cope with the evolution of the Grid API, or to support migration to a different Grid platform. However, separating Grid-specific concerns from other functional and non-functional concerns in the source code for a Grid application is not a trivial task. This happens because the abstractions provided by the Grid are often used as the mechanism to decompose the Grid application. As a consequence, the application's functional code becomes tangled with the Grid-related code, with the latter being scattered across several source code modules [14].

In this paper, we show how to explicitly separate crosscutting Grid concerns in a parallel Java application. This process, named *GridAspecting*, uses (i) a restricted subset of the Java threads model as the mechanism for application decomposition, and (ii) aspect-oriented programming (AOP) [8] for allowing parallel execution of the application's threads as Grid tasks. More specifically, the process uses an AOP language, i.e. AspectJ [1], to dynamically intercept all thread creation and thread initialization calls issued by the Grid application, and to replace them with calls to the corresponding task creation and task initialization services provided by the Grid API. This way, all Grid-related code is encapsulated in aspects, thus improving the application's modularity. In addition, by relying on Java's native concurrency abstractions, which are likely to be more familiar to Java programmers than a specific Grid API, the process simplifies the Grid programming model and makes it possible to test a Grid application (functionally at least) concurrently, even without the Grid. This last feature is particularly useful for avoiding the relatively long submission-execution-evaluation cycle that is typical to the development of Grid applications[H1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'06, April 23–27, 2006, Dijon, France.

Copyright 2006 ACM 1-59593-108-2/06/0004...\$5.00.



The rest of the paper is organized as follows: section 2 gives an overview of the Grid programming model, including an example using a specific Grid platform; section 3 presents the GridAspecting process in more details, section 4 evaluates the GridAspecting process by employing it in a real system; section 5 covers some related work, and, finally, section 6 concludes the paper with a summary of our results and by pointing out directions for future work.

## 2. GRID PROGRAMMING

A Grid can be defined as a hardware or software infrastructure capable of providing new functionalities by the aggregation of existing resources and components [5]. This infrastructure should support applications that involve the manipulation of a large amount of information and a high demand for processing power, such as it is typical in knowledge management and data mining applications [6].

The services and tools provided by most Grid platforms can be used either interactively, by means of a command line interface, or via an application programming interface (API). Figure 1(a) shows part of the code for the method *sort* of the class *GridSorter*, which implements a parallel version of the MergeSort algorithm using the API of OurGrid [11]. This method receives as an invocation parameter an array of strings whose elements are subsequently divided evenly in as many files as indicated by the number of desired tasks (lines 13-23). Then, each generated file is encapsulated in the form of a *TaskSpec* object (Figure 1(b)). Once all tasks have been created, they are grouped into a *JobSpec* object and submitted to the Grid for parallel execution (lines 24-27 in Figure 1(a)). After all sorted files have been received back from the Grid, they are merged into a new array that is then returned to the caller. Notice that the code that uses the OurGrid API (shown in bold face[H2]) is clearly tangled with the code responsible for decomposing the application into tasks in Figure 1.

Figure 1(b) shows the code for the method *createSorterTaskSpec*, which is used to create a Grid task using the OurGrid API. To create a new task, it is necessary to inform OurGrid which files should be sent to each Grid machine (lines 2-3). These machines will be responsible for the parallel execution of each of the tasks

submitted to the Grid. In the example, it is only necessary to send the file *inputFile* to the Grid machines. The tasks themselves correspond to simple executions of the *sort* command (line 4) typically found in UNIX systems. It is also necessary to indicate where the remote files should be stored in the local computer (lines 5-6). Such information is also included in the *TaskSpec* object that is returned by the *createSorterTaskSpec* method (line 9). For space reasons, all *try/catch* blocks and code lines that did not use the OurGrid API were excluded from the example[H3].

## 3. THE GRIDASPECTING PROCESS

In order to facilitate the development and testing of grid-based applications, we have developed a process named GridAspecting, which offers implementation guidelines to assist developers in designing applications so that they can be executed in a grid or tested in a local machine. More specifically, because most Grid middleware platforms are written in Java, the GridAspecting process helps a Java programmer familiar with Java's concurrent programming model in writing Grid-based parallel applications through the use of AOP. As a result of the process, the parallel execution of the application may be simulated locally, using threads, with all Grid-specific code being completely encapsulated in aspects.

The process emerged from our experience in applying AOP to separate Grid-specific concerns in parallel Java applications of different problem domains. It encompasses four main steps, as described below.

### 3.1 Process Steps

#### *Step 1: Task identification and separation of Grid concerns*

When the process is applied to help the parallelization of a sequential application, the first step is to identify potential task candidates in the application source code. In the case which it is applied to help restructuring an existing parallel application whose task boundaries are expected to be explicitly defined, the first step is to localize and explicitly separate all code constructs related to use and configuration of the underlying Grid platform. The first case (i.e., writing an application) developers do not need to know

details about grid; it is only required a deep knowledge of the application's problem domain. However, in the second case developers need to understand details about the grid, that is, the resources and services provided by the Grid API. In both cases, programmers can benefit from existing approaches for concern identification, such as those described in [10]. The process doesn't give any guideline to identify concerns, and this task is left to the programmer. [H4]

### Step 2: Task implementation

The aim of this second step is to (re)implement the application's tasks identified in Step 1 as subclasses of *Thread*. Thus, it allows the application to be executed concurrently, regardless the presence of the Grid. The *run* method of each *Thread* subclass will contain only the functional code identified for each task, without making any direct reference to the Grid API. Moreover, any form of data communication from the main application to its (created) task objects should be implemented exclusively via parameters passed to the task constructor. There are further restrictions that must be followed in this step. These will be discussed in section 3.2.

### Step 3: Task execution and coordination

In this third step the programmer must implement the part of the application responsible for task execution and coordination, which comprises creating and executing the tasks (initially implemented as thread objects) and collecting and processing their results. The tasks should be initialized by creating objects of the task classes implemented in Step 2, and by calling their *start* method. Once the tasks have started, the application should wait for their conclusion before collecting the results. To do this, the application must call the *join* method of each task object, which suspends the application's main thread until all tasks have finished their execution. At this point, the application can safely collect and process the results produced by each task.

At the end of this step, the application is functionally

operational and can be tested concurrently, without the need to have a Grid platform. Figure 2 shows the code for the methods *sort* and *createSorterTaskspec* of the class *SorterThread* and the class *SorterThread*, which was implemented as the result of applying the first three steps of GridAspecting to the class *GridSorter* shown in Figure 1.

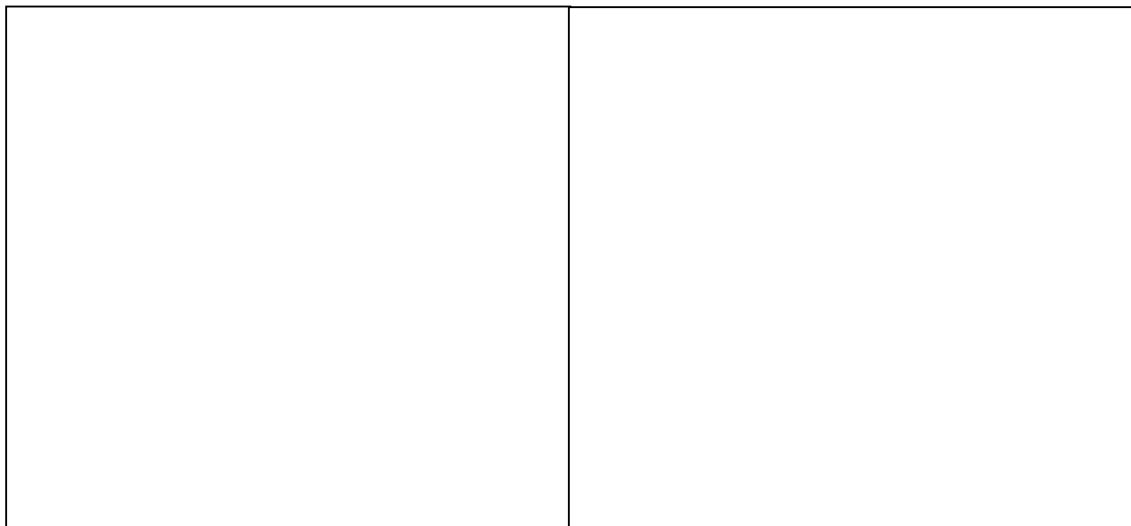
### Step 4: Task parallelization using aspects

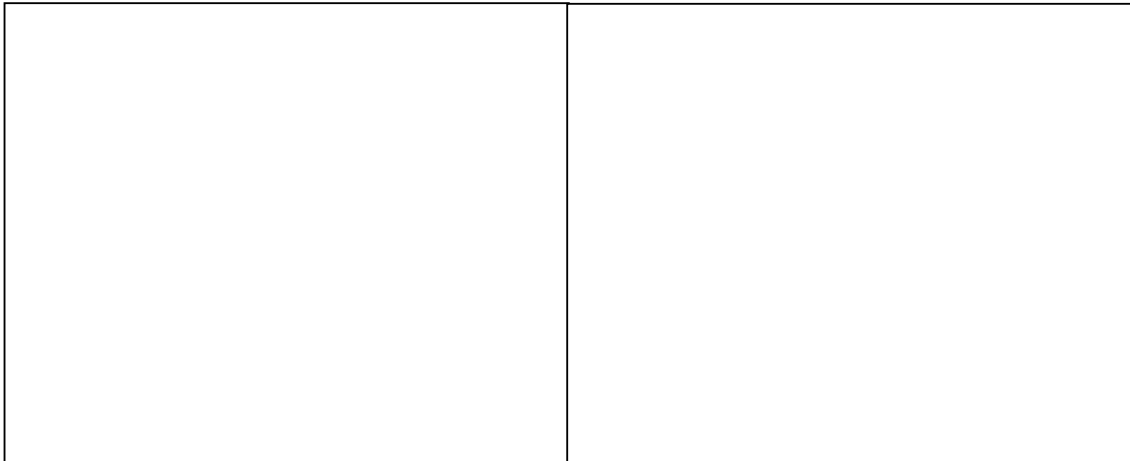
The last step of the process consists of enabling the concurrent application to be executed in parallel, using a Grid. To this end, each task object created in Step 3 must be serialized along with all the required resources (e.g. compiled code, data files, etc.) prior to initialization, in order to be submitted to the Grid for parallel execution. The code for serialization of task objects and their subsequent submission to the Grid is implemented entirely using aspects. The application programmer must implement two types of aspects for each task subclass implemented in Step 2: a *task initialization aspect* and a *task execution aspect*.

#### i) Task initialization aspect

This type of aspect is used to intercept the initialization of each task object created in Step 3. It has two main goals: to allow a given task object to be serialized and stored locally, and to turn the task class into an executable class.

The first goal is achieved by extending the task class so that it implements the interface *Serializable*. This is done using the *declare* clause of AspectJ [1]. To achieve the second goal, the task class must be extended so as to define a *main* method. The implementation of the *main* method must include the code for unpacking the serialized task object sent to the Grid, executing the object's *run* method, and serializing the object again (along with any pertinent result stored as an instance attribute) in order to send it back to the original machine. In the case that the results are stored in an external file, information about that file such as name and path location must also be kept in an instance attribute, so that it can be accessed by the main application in the original machine.





## ii) Task execution aspect

This second type of aspect is used to intercept the execution flow of the application in two moments: at task execution time and also at the time of collecting the results (as described in Step 3). The definition of the pointcuts for the aspect should comprise join points of type method call matching both the *start* and *join* methods of each task class.

For each type of join point, the programmer must implement an advice (using AspectJ's *around* clause) in order to intercept the execution of the target method and to replace it by the method defined in the advice body. In the advice for the *start* method, the programmer must implement the code to serialize the target task object, to create a new Grid task, and, after all tasks have been created, to submit them to the Grid using the appropriate Grid API. When the process is being applied for restructuring an existing Grid application, programmers use this step to reuse the code constructs related to Grid use that were identified in Step 1.

As we have mentioned, after a Grid task has been executed, its results can either be kept in instance attributes of its local task object or stored in an external file in the remote Grid machine. For the former, the advice for the *join* method must include the code to unpack the appropriate attribute values contained in the serialized task object received back from the Grid, and to assign them to the same attributes of the corresponding task object originally created by the main application at the local machine. In the case that the tasks results are stored in an external file, it suffices for the advice to check whether the Grid has already transferred the corresponding data back to the local machine. Figure 3 shows part of the advice implemented for the methods *start* and *join* of the *SorterThread* class shown in Figure 2.

## 3.2 Implementation Restrictions

Despite being relatively simple, the proposed process requires that all task classes implemented in Step 2 follow very strict rules. These rules are necessary to guarantee that the task objects can be safely converted to proper Grid tasks by the aspects. The rules are: (i) the task class must not define a *main* method, as this will be introduced later by the initialization aspect; (ii) a task object must not access any static variable of its own class or of any of its

ancestors, since the tasks will be executed independently by the Grid in potentially distributed machines; (iii) a task class must store its results in its own instance attributes or in a single external file, since those are the only means allowed for communication between the tasks submitted to the Grid and the main application; (iv) finally, the task class must not access any other external file unless its absolute path name has been specified as an invocation parameter to the class constructor.

Note that this approach can be easily applied in Bag-of-Tasks (BoT) applications, which are those applications whose tasks do not communicate during execution. Despite their simplicity, BoT applications are used in a variety of scenarios, including data mining, massive searches (such as key breaking), parameter sweeps, simulations, fractal calculations, computational biology, and computer imaging [2]. Currently we are working for applying the process to applications with communicating tasks.

## 4. A CASE STUDY<sup>[H5]</sup>

ExpertCop [9] is a multi-agent simulation tool that uses a genetic algorithm to analyze the performance of police routes. The results of each simulation feed the genetic algorithm, which aims at optimizing the routes in terms of reducing the number of crimes. As this approach requires a large number of simulations, it was necessary to improve the performance of the tool. Thus, we decided to parallelize ExpertCop using OurGrid [11], which led us to apply the GridAspecting process to its source code.

Since the original application did not use any Grid resource, in Step 1 we identified the parts of the simulation code amenable for parallelization. In Step 2, we implemented a *Thread* subclass, named *CromossomoThread*, to carry out multiple simulations in parallel. As each simulation returns the number of committed crimes, we added a new attribute to *CromossomoThread* to store this result. The parameters necessary for the simulation were passed as invocation parameters to the class constructor, as recommended by the process, which were then assigned to local instance variables.

In Step 3, we replaced the calls to the simulation routines in the original code by the creation and subsequent initialization of *CromossomoThread* objects. Then we called the *join* method of each created task object so that, after all tasks had been executed, their results could be collected by accessing the appropriate attribute. As expected, once Step 3 was completed, we were able to run the application concurrently, without the Grid.

In the last step, we implemented the aspects responsible for submitting the *CromossomoThread* objects to the Grid. In the task initialization aspect we followed the approach of unpacking the serialized task object, invoking its *run* method and, as the task result is stored in a local instance attribute, serializing the object again so that it could be returned to the original machine. The pointcut definition for the aspect's *start* advice resembled that shown in Figura 2(b), but using the *CromossomoThread* class as target instead of *SorterThread*. In the advice for the *join* method we assigned the result value contained in the serialized task object returned by the Grid, to the same attribute of the corresponding task object in the main application. Finally, we tested the application again, this time with the aspects enabled. Once more it ran successfully, but now showing a significant performance gain (in the order of hours of execution) compared to the original implementation. Nonetheless, a deeper evaluation of the aspect-oriented re-implementation of the system to the Grid is still necessary in order to better assess its gains in terms of both modularity, easiness of use and performance.[H6]

## 5. RELATED WORK

We are unaware, at the time of writing, of any other aspect-oriented approach specifically tailored for implementing Grid applications. However, our work presents similarities with a number of other researches, as we discuss below.

In [12], the authors describe an aspect-oriented code restructuring process, named *Aspecting*, which can assist a software engineer in identifying crosscutting concerns in an existing object-oriented Java application. They also provide some guidelines on how the identified concerns could be explicitly separated from the application code using an AOP language such as AspectJ. Although we share some of the same interests (e.g. identifying crosscutting concerns in an existing system), in addition to the name *Aspecting*, that work differs from ours in that it aims at supporting the identification of general crosscutting concerns, while we focus specifically in identifying Grid-related concerns. Moreover, in our work the non-aspect based version of the application is already functionally equivalent to the version that is produced by enabling the aspects, which is not the case with the implementation that is produced using *Aspecting*.

In [13], it is shown how a distributed Java application, implemented using RMI, can be converted to a functionally equivalent application that can run locally, with the distribution code (as well as the code responsible for the persistence mechanism) being encapsulated in aspects. That work resembles our work in that both versions of the target application, i.e. with and without aspects, would also be functionally equivalent. Nevertheless, while that work uses AOP to separate distribution and persistence concerns from the application code, our interest is in using aspects to allow both concurrent and parallel executions of a Grid application, without exposing the underlying Grid API.

The development of Grid applications using special Grid services provided in the form of a reflexive middleware has been discussed

in [14]. With this approach, a Grid application would be written as a traditional sequential application, and it would be up to the reflection mechanism to automatically decompose the application and submit its decomposed parts to the Grid. That work shares our aim of making the use and configuration of the Grid transparent to the application programmer. However, it differs from ours in the way to achieve such transparency. While it advocates using the reflection mechanism to completely hide the application decomposition mechanism from the programmer – a quite bold goal, as the authors themselves admit in [14] – we search for transparency in a more pragmatic way, by combining threads and AOP in a way to offer a Grid programming model that is independent of any specific Grid technology.

Another recent work [3] describes how AOP was used to facilitate the reorganization and evolution of an existing Grid platform. Like our work, that work also uses aspects to intercept the execution of thread objects. However, in that work aspects are used primarily to improve the internal structure of the Grid, by explicitly separating thread management concerns from the rest of the system implementation. In our work, in contrast, we use AOP as a way to decouple the application decomposition mechanism from the underlying Grid platform.

## 6. CONCLUSION

This paper presented an aspect-oriented implementation process for explicitly separating crosscutting Grid concerns in Grid-based Java applications. The process offers a number of benefits to Grid application programmers, including: better modularity, with all Grid-related code being encapsulated in aspects; a simplified programming model, based on threads; and the possibility to test a Grid application concurrently, in a single machine. The process was illustrated using a simple example and further validated through a more detailed case study involving a real application.

As future work, we are looking into some aspect-oriented metrics that could help us in quantitatively evaluating the modularity gains achieved with the process. We are also working on the generalization of the task initialization and task execution aspects, so as to increase reuse and, consequently, to reduce the effort necessary to apply the process to new development scenarios. Another interesting line of research is to investigate how to make the implementation restrictions (explained in section 3.2) explicit, so as to facilitate their verification (perhaps automatically) against the source code for an existing concurrent application.

## 7. REFERENCES

- [1] AspectJ Project. AspectJ Home Page. Available at <http://eclipse.org/aspectj/>. Accessed on 12/07/2005.
- [2] Cirne, W., Brasileiro, F., Andrade, N., Santos, R., Andrade, A., Novaes, R., Mowbray, M. Labs of the World, Unite!!! Universidade Federal de Campina Grande, Departamento de Sistemas e Computação, UFCG/DSC Technical Report 01/2005. Available at <http://walfredo.dsc.ufcg.edu.br/papers/Labs%20of%20the%20World%20Unite%20v12.pdf>.
- [3] Dantas, A., Cirne, W., Saikoski, K. Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society (JBCS)*, 11, 3 (2006). To appear.

- [4] Foster, I., Kesselman, C., Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15, 3 (2001).
- [5] Foster, I., Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure*. Kaufmann Publishers, Orlando, FL, 1999.
- [6] Sousa, F., Ayres, L., Furtado, V. A Knowledge-Based Architecture for Helping in the Optimization and Development of Data Mining Applications in Grids. In *Proc. of the AAAI'05 Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing* (Pittsburgh, Pennsylvania, USA, July 2005).
- [7] Globus Project. Globus Home Page. Available at <http://www.globus.org>. Accessed on 12/07/2005.
- [8] Kiczales, G. J., Mendhekar, L. A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. Aspect-Oriented Programming. In *Proc. of the 11th Eur. Conf. on Object-Oriented Programming (ECOOP'97)* (Jyväskylä, Finland, June 1997). Springer-Verlag, LNCS Vol. 1241, 220-242.
- [9] Melo, A., Belchior, M., Furtado, V. Analyzing Police Patrol Routes with the Simulation of the Physical Reorganization of Agents. In *Proc. of the 6th Int. Workshop on Multi-Agent Based Simulation (MABS'05)* (Utrecht, The Netherlands, July 2005). Springer-Verlag.
- [10] Murphy, G. C., Lai, A., Walker, R. J., Robillard, M. P. Separating Features in Source Code: An Exploratory Study. In *Proc. of the Int. Conf. on Software Engineering (ICSE 2001)* (Toronto, Ontario, Canada, May 2001).
- [11] OurGrid Project. OurGrid Home Page. Available at <http://www.ourgrid.org>. Accessed on 12/07/2005.
- [12] Ramos, R. A., Penteadó, R., Masiero, P. C. A Code Restructuring Process Based on Aspects. In *Proc. of the 18th Brazilian Symposium on Software Engineering (SBES 2004)* (Brasília, DF, Brazil, Oct. 2004). In Portuguese.
- [13] Soares, S., Laureano, E., Borba, P. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. of the 17th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)* (Seattle, Washington, USA, Nov. 2002).
- [14] Tramontana, E., Welch, I. Reflections on Programming with Grid Toolkits. In *Proc. of the ECOOP'04 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)* (Oslo, Norway, June 2004).