Project No. FP6-004265

CoreGRID

European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

Network of Excellence

GRID-based Systems for solving complex problems

Deliverable D.PM.07 – Innovative Features of GCM (with sample case studies): a Technical Survey

Due date of deliverable: 31 May 2007
Actual submission date: 12 September 2007

Start date of project: 1 September 2004 Duration: 48 months

RESPONSIBLE PARTNER: INRIA

Revision: *Draft*

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: programming model, components, grid, high performance, scalability

Contents

1 Executive Summary

This deliverable is intended to show what are the highly innovative features of the GCM, and how they can be used to address challenges raised by the Grid applications. It illustrates the innovative aspect of the GCM by presenting several innovative and effective Grid developments that have been realized around the GCM initiative.

1.1 Recalling GCM Principles and Central Features

First, the GCM proposes a solid and adequate parallel and distributed programming model laying the foundation for building any form of grid application. Its qualities must be those of expressiveness, extensibility, solid theoretical foundation for a clean semantic and capability to reason upon, suitability for optimization and competitive implementations.

The crucial features of the GCM are:

- *Fractal as the basis for the component architecture:* The main characteristics we benefit from Fractal are its hierarchical structure, the enforcement of separation of concerns, its extensibility, and the separation between interfaces and implementation.
- Communication Semantics: GCM components should allow for any kind of communication semantics (e.g., streaming, file transfer, event-based) either synchronous or asynchronous. Of course, for dealing with latency, asynchronous communications will probably be preferred by most GCM frameworks.
- Support for deployment: distributed components need to be deployed over various heterogeneous systems. The GCM defines deployment primitives for this.
- Support for many-to-many communications: often, Grid applications consist of a lot of similar components that can be addressed as a group, and that can communicate together in a very structured way. The GCM also intends at providing high level primitives for a better design and implementation of such *collective communications*.
- Support for non-functional adaptivity and autonomic computation: Finally, the Grid is an highly evolving environment; and Grid applications must be able to adapt to those changing runtime conditions. For this we propose to allow for both reconfiguration of the component control aspects, and autonomic computation support.

Rather than presenting again those features as already shown in previous deliverables of the programming model institute [33, 23], this deliverable takes some selected key functionalities for a Grid application that have been implemented thanks to the GCM, or within the context of the GCM. The following of this extended summary will overview those features, providing a short summary, focusing on the innovative aspects of the features presented in this paper, and on the impact of the core of the GCM on those high-level features.

In other words, this deliverable report experiences built on top of the GCM specification. Each of them is closely related to the GCM component model and somehow benefits from its features, as explained below.

1.2 Composing Grid Applications

This deliverable starts by a strong emphasis on the importance of hierarchical component composition in the design of a component system. Hierarchy is particularly important in a distributed and highly evolving environment. Moreover, hierarchy is a key feature for scalability, and autonomous control of the component systems, which are key concerns in the design of the GCM, and are part of the crucial challenges Grid programming must address.

Section 3 will focus on the hierarchical composition of Grid application and particularly on the graphical design of Grid application, through a dedicated IDE, the Grid Integrated Development Environment: *GIDE*. GIDE supports visual interaction for developing applications. Technically, the overall composition operation within the IDE is enabled by interconnecting three different modules viz. ADL Parser/Verifier, ADL Renderer and ADL/Source Code Generator.

1.3 Towards A High-Level Scripting Approach for Building GCM Component Systems

Whereas Section 3 focuses on a graphical view of the component composition, relying on an ADL (architecture description language) describing an initial component composition in a very static way, leaving space for dynamic creation and reconfiguration of new components, an alternative view consists in using a script-based approach to design a component system.

Section 4 presents a high-level scripting language for programming component applications on the Grid. By using a dynamic interpreted language approach, it is possible to design a flexible and powerful notation, which covers all aspects of deployment, hierarchical and workflow composition, parametrization and configuration of components. The scripting approach can be applied to the process of rapid application development, prototyping and conducting scientific experiments on the Grid. The prototype which was developed demonstrates the feasibility of the proposed solution.

Section 4 will show that the GCM is both adapted to an ADL description of applications like in Fractal [29], but also to a script-based approach for the description of Grid applications

1.4 Support for Master-worker Paradigm in GCM

Section 5 defines an extension of the GCM able to ease the definition and deployment of master/slave applications. This section shows how the GCM can easily be adapted, in the case of a very classical and widely used application scenario, to reduce the definition of the application to a very simple skeleton, and make all the deployment and instanciation process automatic. To summarize the user defines an abstract description of its application together with a communication pattern expressing how data are scattered toward the different workers, and the remaining is done automatically at deployment time.

The proposal to improve the support of the master-worker paradigm in GCM is based on the concept of *collection*. A collection is defined as a set of *exposed* ports, bound to some internal component type ports. A collection behaves like a component: it can be connected to other components. However, such a composition is done in an *abstract* architecture description, which represents the user's view of the application. At deployment time, a collection is turned into a concrete assembly, formed by some internal component instances and by an instance of a *request transport pattern*. A *pattern* represents an implementation of an algorithm that specifies how to transport requests from a *master* to several *worker* components.

Its implementation should be done by experts and it may be based on software components.

Section 5 will underline the importance of multiple components, and of abstraction of ports for such multiple components, multicast and gathercast interfaces defined by the GCM can be considered as the basic blocks allowing to specify all kinds of such collective communications; each communication pattern can indeed be realized through a set of particularized multicast and gathercast interfaces.

1.5 Optimizing group communication for GCM

Section 6 illustrates how group communications, as they required for implementing GCM multicast interfaces can be implemented in an efficient way.

This section shows that the GCM is a good environment and specification for implementing optimized communication patterns. Moreover, some implementations of the GCM can benefit from these optimized group communications in order to provide efficient multicast interfaces.

1.6 The Integrated Toolkit: Supporting Grid Applications

Section 7 presents the Integrated Toolkit that has been specified and designed in the framework of task 7.3 of the CoreGRID Institute on Grid Systems, Tools and Environments has as main objective the specification and design of an *Integrated Toolkit*: a framework which enables the easy development of Grid-unaware applications.

Moreover, the Integrated Toolkit design can also offer an alternative to develop Grid-aware applications. The componentised structure of the Integrated Toolkit makes possible to use it as a whole or to deploy solely specific subcomponents. For instance, a programmer interested in adding a scheduling functionality to an application could choose to use only the Task Scheduler subcomponent of the Integrated Toolkit, binding its interfaces to the ones of the application components. This shows the interest of using a hierarchical component model for programming Grid applications and platforms.

The Integrated Toolkit is of particular interest in this documents as it both illustrates how the GCM can be used to design a Grid platform itself, and also how Grid-unaware components can be integrated in a GCM component system. This section also underlines how the Integrated Toolkit benefits from the GCM features and as such illustrates the impact and advantages of the GCM. Consequently, this section perfectly illustrates the benefits of using the GCM for programming Grid platforms and applications.

The main features of the GCM that are of particular interest in the design of the integrated toolkit are: hierarchical design, synchronous/asynchronous communication patterns, collective communications and deployment support; they will be detailed in Section 7.2.

1.7 Autonomicity in the GCM: Autonomous Support of Grid Applications

Section 8 Describes autonomic behavioural skeleton as an important contribution to the autonomic management of GCM components.

Behavioural skeletons preset a compromise: being skeletons they support reuse, while their parametrization allows the controlled adaptivity needed to achieve dynamic adjustment of QoS while preserving functionality. Self-management of a basic set of skeletons (farm, data-parallel, and active-replication) will be defined, together with the GCM implementation of on of those class: the farm skeleton.

Section 8 illustrates how behavioural skeletons can be applied and implemented within the GCM component model. This contribution entails, relatively to the GCM characteristics:

- Designing autonomic algorithms that deal with hierarchical components, and benefit from the hierarchical structure of components;
- Better supporting stream-based communications;
- Specifying a new kind of multicast interfaces, called *unicast interfaces*.

Such a framework for autonomic components strongly benefit from the GCM component definition and component reconfiguration facilities.

2 Introduction

The GCM (Grid Component model) addresses the novel characteristic challenges of Grid computing - viz. heterogeneity, large-scale distribution and dynamicity - in terms of programmability, interoperability, code reuse and efficiency. GCM mainly focuses on the programmability of Grid applications, but keeping in mind the other challenges. Programmability deals with the expressive power of the language mechanisms that are offered to the programmers, and what is the burden for them to effectively use those mechanisms. A short overview of current proposed grid frameworks makes us believe that it is in the programmability dimension that resides the greatest divergence between those solutions. Schematically, these solutions range

- from low-level message-passing (e.g. MPI), RPC or RMI based traditional parallel and distributed programming models – simply ported to tackle grid issues – by which the program itself, dictates and orchestrate the parallelism and distribution of computing and communicating entities;
- to solutions in which the orchestration or choregraphy of the set of parallel and distributed entities is guided from the extern of these entities, not necessarily in a centralized manner (e.g. workflow languages [32]).

It is our opinion that these two categories are not exclusive, because the spectrum of applications that could benefit from running on grids is not closed at all. The purpose of the GCM is to lie in between those two extreme points of view: a component approach allows both explicit communications between distributed entities like in MPI, and high-level management of the distribution of those entities and their interactions, like in workflows.

2.1 Components as an Abstraction of the Deployment Unit

A crucial point that drove the design of the GCM is the interaction between components viewed as a software entity and viewed as the unit of deployment and parallelism.

A general issue when designing a component model is the advised granularity of the components: “what is the size of a component?”. In the case of a hierarchical component model like Fractal, this question becomes “what is the size of a primitive component?”. Fractal does not precise any granularity for the components, but the existence of composite bindings and some of the features of the model suggests a rather fine grained implementation: a primitive component should contain a few objects.

When addressing distribution aspects of a component model, the same question arises again, but becomes more complex: “what is the relation between the unit of composition (the primitive component) and the unit of distribution?”. Like Fractal, the GCM does not enforce precisely any granularity of the component systems. However, in order to allow GCM primitive components to be the unit of distribution for a GCM implementation, we consider that GCM component implementations would probably have a coarser granularity than Fractal ones. This difference of granularity between Fractal and the GCM partially explains why some of the features that could be implemented by a small Fractal component and are highly used in a Grid setting have been defined as first class citizens in the GCM. For example, multicast interfaces could be express in Fractal by binding components that perform the broadcast, but such components would be too small to be used as the unit of distribution.

Compared to other component models, the GCM has been conceived with a granularity that is somehow in middle between small grain Fractal components and

very coarse grain component models, like CCM where a component is of a size comparable to an application. Somehow, GCM has been conceived thinking of components of the size of an MPI process, though it can be used in a much finer or coarser grain way.

This deliverable takes some selected key functionalities for a Grid application that have been implemented thanks to the GCM, or within the context of the GCM. These features are:

- The hierarchical composition of Grid applications, either as an architecture description language, or graphically through an IDE, or even via a scripting language (Sections 3 and 4);
- Special support for the design of master-worker application, in order to better conceive the most classical Grid applications (Section 5);
- Support for various communication patterns and semantics, and their optimization, including as an illustrative example, optimized group communications (Section 6);
- Design of a Grid toolkit environment as a GCM component system, allowing both to prove the adequacy of the GCM, and to benefit from GCM features, e.g. reconfiguration, deployment, optimized communications, ... finally leading to a unique dynamically evolving, highly structured, distributed Grid integrated toolkit (Section 7);
- Support for autonomic behavioural skeletons able to adapt themselves to a constantly evolving environment and requirements in terms of quality of services (Section 8).

3 Composing Grid Applications

This section aims at emphasizing the importance of the hierarchical design of a component system, and presenting some tools supporting development of such hierarchical components. The IDE presented in this section was developed in the WP4 of the GridCOMP European project.

3.1 Introduction

Grid systems have become tightly integrated as an indispensable part of the computing community. Applications from different domains use computational Grids and although they offer remarkable benefits for a given problem, the actual investment in terms of time includes both the running time and the time for developing the solution. This implies that development experience has a direct impact on the "time-to-solution" issue. The development process can be simplified by following a component-oriented development paradigm [48].Although there exists substantial amount of ground work in facilitating the development for and the utilisation of modern Grid systems, rarely do any of them offer a unified and integrated solution. There is also no support for full-fledged component-oriented development. Here, we present the architecture and design of a component-based Grid integrated development environment (GIDE), particularly focusing on the composition process. Our design supports component-oriented development and post-devel opment functionalities such as deployment, monitoring, and steering. These functionalities target different user groups of the Grid developers, application users and data-centre operators.

3.2 The IDE

Figure 8 below shows a perspective for the integrated environment consisting of a number of views supporting these functionalities, namely component composition, resource monitoring, deployment and active-object monitoring.

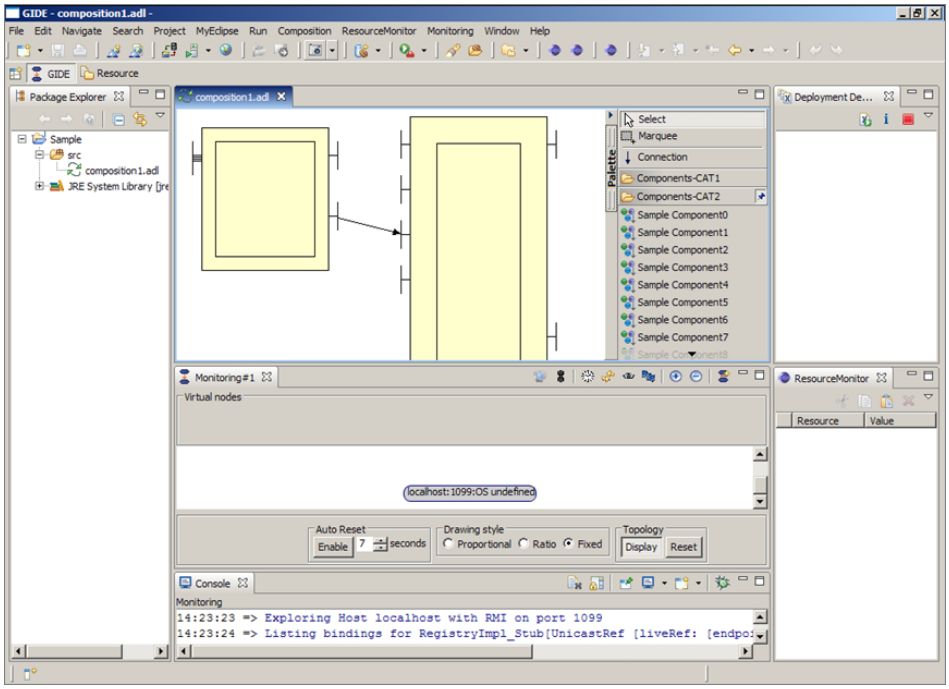

Figure 1: The Integrated Development Environment for Grid

Within our GIDE, the fully interactive environment is built using the underpinning Graphical Editing Framework (GEF) [31], which provides a framework for creating model-agnostic interactive applications within Eclipse [30]. The Graphical Editing Framework (GEF) and Eclipse facilitates handling of different events arising within different views of the IDE. Events are captured through a message loop processed by the Eclipse Platform, which are then routed to the event handlers within the plugin. These event handlers are model-agnostic so that the changes are reflected upon editing. The GIDE componentises associated operations and elements so that it could easily fit within the domain of Eclipse. Figure 2 shows our current model of representing components and compositions

3.3 Composition

To successfully support and provide a development platform posts the following requirements:

- Provide an integrated programming and composing GUI.
- Provide facilities to bind both normal code and legacy code into primitive components.
- Tools for assembling existing Grid components and larger composite components. (Preferably graphical)
- Tools to finalise the configuration of the application before execution.

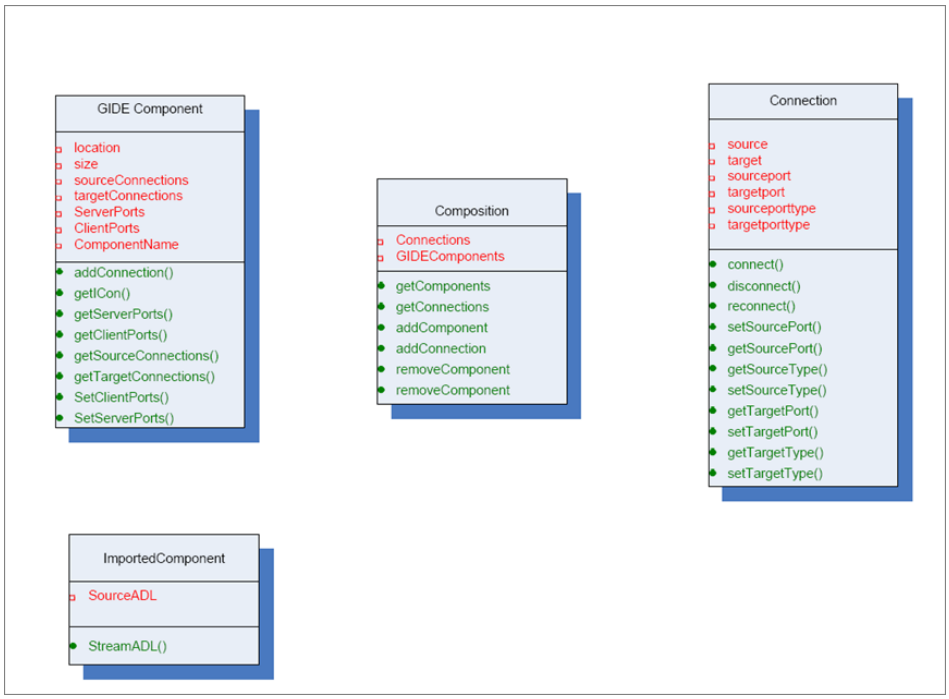

Figure 2: Component Composition Model

- Tools to search for suitable components.

As discussed in Section 3.1, in a component-oriented development environment, applications are made up of several components. One potential advantage of using the GIDE for that process is the support for visual interaction for developing applications. The central area focuses the user on the graphical composition view which provides the developer with a palette of available components that can be dragged and dropped on the composition canvas. Components can then be resized and moved. It is our vision to support fully customisable editing features including adding/removing connections between ports of different components, deriving compositions by drawing a membrane around the components, full context menu support, full context-aware editing support for ADL files, importing and exporting ADL files, and necessary source code/ADL code generation. In addition to this, the composition editor supports code-view for graphical view for ADL files. Once an ADL file is imported, the source is also imported into the solution which the developer works on. This way the developer will be able to switch between the graphical view and a view of the fractal description of the component as an ADL file. The ADL file can then be exported and used for deployment. In a typical scenario, the developer would begin the process simply by launching eclipse within which the GIDE will be included, given that it was pre-installed. Following this, the development process would be as usual within Eclipse. Figure 3 below shows the overall user interface for the GIDE Composition Editor. When composing an application, users can drag and drop different components from the components gallery, which is built at runtime. Their properties could be changed using the properties tab. As they are dragged and dropped, necessary skeleton codes are generated in the back-end and they can be viewed or edited as necessary using the Solution/File Explorer. Additional information about each component is available in the metadata tab. From time to time, it may be necessary to import ADL files into an existing project and then to modify the imported file. The GIDE natively supports

importing ADL files within projects. Imported artefacts will appear as part of the solution and can either be edited or rendered on screen for further editing through the context menu.

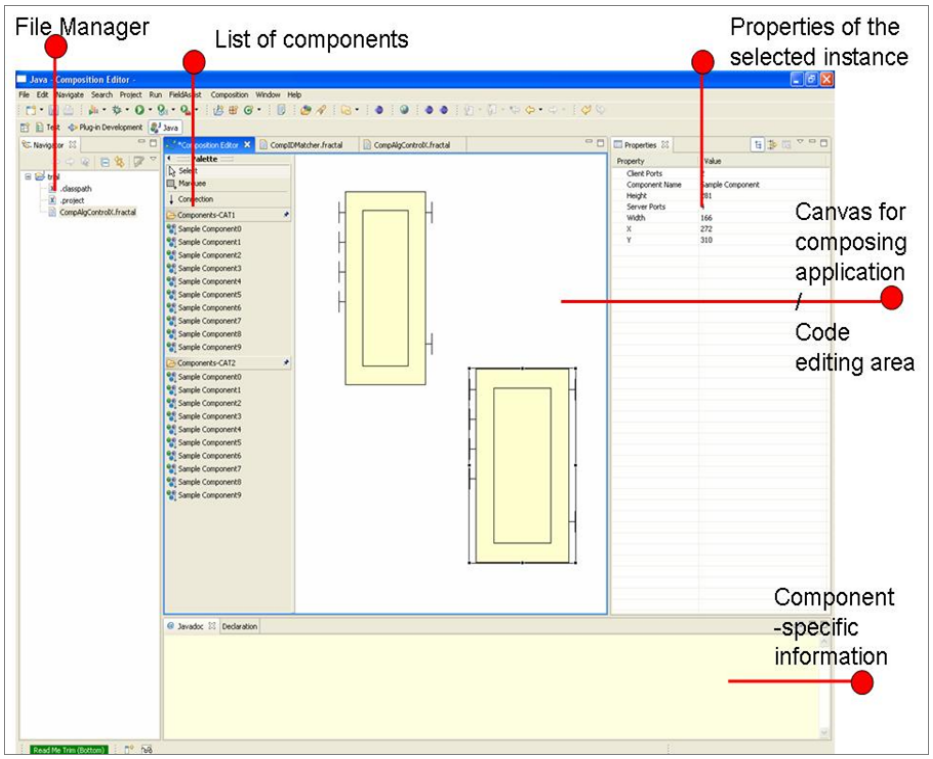

Figure 3: User Interface of the Composition Editor

The overall composition operation within the IDE is enabled by interconnecting three different modules viz. ADL Parser/Verifier, ADL Renderer and ADL/Source Code Generator. The ADL/Parser module parses and verifies the ADL files before associating them with the project. The renderer relies on the GEF for rendering an ADL file content. Finally, the ADL/Code generator is responsible for generating or exporting ADL files during or after the graphical composition. The underlying model of the GIDE composition editor is derived from Plain Old Java Objects instead of deriving the model through the Eclipse Modelling Framework (EMF) . Although relying on EMF could have given potential gains in automating the process, we found the approach of mapping our plain java-based model to GEF attractive as we could easily re-use some of the mappings, in particular GCM components.

4 Towards A High-Level Scripting Approach for Building GCM Component Systems

This section describes a high-Level scripting language for a distribute component model like the GCM. Up to now, the only implementation of the language has been realized in the MOCCA framework, but an implementation allowing to build GCM component systems is envisioned, and would rely on the same principles.

4.1 High-level Scripting With GridSpace - concept and prototype

4.1.1 Introduction

There are two important models of component composition: composition in space and composition in time, which can be relevant to Grid applications [47, 37]. Composition in space, also known as hierarchical composition, involves direct connections between component ports, while a control and a data flow passes directly between connected components. Composition in time assumes that the components do not have to be directly connected, but their server interfaces can be invoked by a client, which coordinates the whole application. In this case both control and data flow pass through the client, which can be a specific application or a more generic workflow engine [39].

There is a need for a high-level programming approach which would enable combining both types of component composition in a way which is flexible and convenient for a programmer. The approach should not be limited to a single component model, since there are many available for programming Grid applications [34]. Moreover, being focused on the Grid environment, it should allow hiding the complexity of underlying infrastructure, automating the process of component deployment and resource selection where possible. It would be also valuable, if the solution could facilitate such aspects as component configuration and passing parameters to the application.

Here, we describe a top-down approach to solving the problem of component composition on the Grid. The proposed solution is based on a dynamic scripting language [59]. This solution is especially well suited for rapid application development (RAD), prototyping and experimenting scenarios.

There are several ways of component composition: low-level API, scripting languages, descriptor based programming (ADL), skeletons and high order components, and graphical tools.

Each component standard, such as Common Component Architecture (CCA) or Fractal provides an API (possibly in many programming languages) to perform basic operations on components. This API is then used by other high-level interfaces, facilitating the composition process.

One common approach for component composition is to use a scripting language. Some frameworks define their specific notation, as in the case of CCAFFEINE framework [6], others offer direct interface from a script to the framework API. The latter case is realized in XCAT [43] and MOCCA [46], where applications can be assembled using a script written in Jython [41] or JRuby [40]. These languages have been selected partially because they use Java-based interpreters and allow seamless integration with Java client-side libraries or component frameworks. When using such scripts, it is possible to combine composition in time with composition in space, since both Python and Ruby are powerful programming languages which allow expressing the control flow and the sophisticated logic of any application.

For composition in space, it is possible to use a Architecture Description Language (ADL). Such a notation, which is present in the component standards such as Fractal [15] or CORBA Component Model (CCM) [20] allows specifying the application structure in the form of a graph showing the connections between components. By introducing a concept of virtual nodes in ProActive and in GCM [53], it is possible to separate the architecture description from the deployment information, which is then provided in auxiliary deployment descriptor files. ADL approach can be supported by graphical tools, however, it is limited in describing dynamic application behavior and does not allow composition in time.

For composition in time, there are specific notations available, called workflow

languages, which specify application flow (control or data) in the form of a graph. There are many workflow systems available for the Grid, such as Kepler [7], Triana [60], Pegasus [26] and K-WfGrid [39] systems. They enable editing the workflow using graphical tools and support specific constructs such as loops, conditions, parallel execution, etc. They are intended to assist non-programmer users in developing applications, however in the case of workflows with many components and complex interactions, they can also become difficult to use. It is also possible to express workflow in an imperative language, as in Grid Superscalar [57].

4.1.2 Concept of a High-level Scripting Language

The concept of our approach to composition of component-based grid applications is based on using a dynamic scripting language. Such a language allows us to design a high-level API for application composition and deployment, which enables specifying the application structure in a concise way. Modern scripting languages allow the programmer to specify the same functionality with considerably smaller number of lines of code than e.g. Java, making the code more readable and thus less error-prone. On the other hand, they provide a full expressiveness needed to specify application behavior in more flexible way than any workflow notation. After careful analysis of possible candidates, we have selected Ruby [55] which is object oriented, dynamic scripting language with a clear and powerful syntax. As an interpreter we chose JRuby [40] which is implemented in pure Java and allows seamless integration with all available Java libraries.

To illustrate the concept of a script, let us consider a simplified application, as shown in Fig. 4. There are three components: the first *Generator* for preparing initial data, *Simulation* performing some computations, and *Output* responsible for storing the results. Such an application can be modelled either using direct connections between components, or as a workflow which is coordinated by external entity, labeled as *RuntimeSystem*.

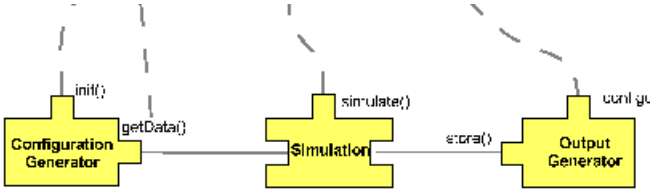

Figure 4: Example component application

Composition support A script allows to easily express both types of composition (see Fig. 5 and 6), while preserving the clear and concise notation. As Ruby is an object oriented language, component instances in the script are represented as objects. Taking advantage of dynamic method definition and invocation it is possible to refer to their ports and port operations using a single method. By using simple loops it is possible e.g. to create collections of components and then iterate over them or connect them in required topologies, such as graphs or meshes.

Deployment specification A programmer assembling a Grid application should have a flexibility in deciding how detailed information about deployment should

```
generator = GS.create("org.example.ConfigurationGenerator")
simulation = GS.create("org.example.Simulation")
output = GS.create("org.example.OutputGenerator")

simulation.inputPort.connect(generator.dataPort)
simulation.outputPort.connect(output.outputPort)

generator.init(steps, size)

simulator.simulate()
```

Figure 5: Example script - composition in space

```
generator = GS.create("org.example.ConfigurationGenerator")
simulation = GS.create("org.example.Simulation")
output = GS.create("org.example.OutputGenerator")

generator.init(steps, size)

for i in (1..steps)
    data = generator.getData(i)
    result = simulator.simulate(data)
    output.store(result)
end
```

Figure 6: Example script - composition in time

be provided manually, and which decisions could be left for automatic tools. We consider three levels of detail:

- Fully automatic: a programmer specifies only the class of a component to create. The location for component deployment is determined automatically by the system:
  `GS.create(componentClassName)`
- Using a virtual node: a programmer specifies a virtual node which component should be deployed on:
  `GS.createOnVN(componentClassName, vn)`
- Manual, by specifying a concrete location, e.g.
  `GS.createConcrete(componentClassName, URI)` where URI is the identifier of a concrete component container (e.g. H2O kernel in the case of MOCCA).

All these levels should be supported by the runtime system, and might be combined by a programmer, e.g. to specify a concrete location for a master component and let the system automatically select resources for worker components from available pool.

Framework interoperability The scripting API for composition and deployment of components is neutral with respect to component model used. Script invocations are translated to underlying Fractal, CCA or CCM API. If more than one component model is supported, then it is possible to combine components from different models into a single application. In the case of component workflow, the runtime system is the central point of inter-component communication, so it acts as intermediary passing results from one component invocation to another. It is also possible to integrate other technologies, e.g. Web services or WSRF in such a workflow. In the case of composition in space, when direct links between components are involved, it is necessary to introduce a *glue* layer between the heterogeneous components, to enable invocation translation. As our research on interoperability between GCM and CCA [45] suggests, it is possible to introduce such a generic glue which can bridge components from different models and frameworks.

4.2 Runtime Support

In order to make GScript operational, there is a need to provide a Runtime library, responsible for providing the information on components in use and to hide the complexity of underlying grid infrastructure. The architecture of the Runtime system is shown in Fig. 7. *Registry* is used for storing all technical information about available components, containers and the state of the Grid resources, updated by *Monitoring* system. The *Optimizer* module is responsible for supporting decisions on (automatic) component deployment. The role of optimizer is similar to the *deployment framework* as proposed in [22]. The *Invoker* module transparently performs remote calls on component ports of different frameworks. The invoker has an extensible architecture which allows plugging in adapters responsible for interacting with different technologies.

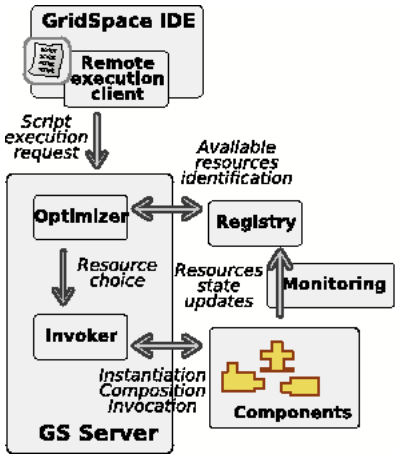

Figure 7: Runtime system of Gridspace

4.3 Development Support

Although programming in a scripting language such as Ruby can be convenient for a programmer, there is always a need to support the development process with user friendly tools which assist in the implementation and help reducing number of mistakes. For GScript, we offer an integrated development environment based on Eclipse platform with Ruby Development Tools enriched by additional plugins. The first plugin is the registry browser which lists all available component classes, their ports and methods. It is connected with a script editor and allows to insert automatically generated code snippets, such as component creation method. Another plugin may be used to browse the component classes categorized using an ontology-based taxonomy. It can be especially useful when searching for a component based on its functionality and to find similar components which are available.

4.4 Prototype of Runtime System

The first prototype of the runtime system has been developed in the scope of ViroLab project where it serves as a Virtual Laboratory runtime. Currently it supports MOCCA components which can be combined in a workflow together with Web services. The prototype has been validated on the sample experiments developed within the virtual laboratory and now together with the script API is in the process of refinement. The registry is available as a Web service, and stores technical information about registered components and services. In the invoker module there are

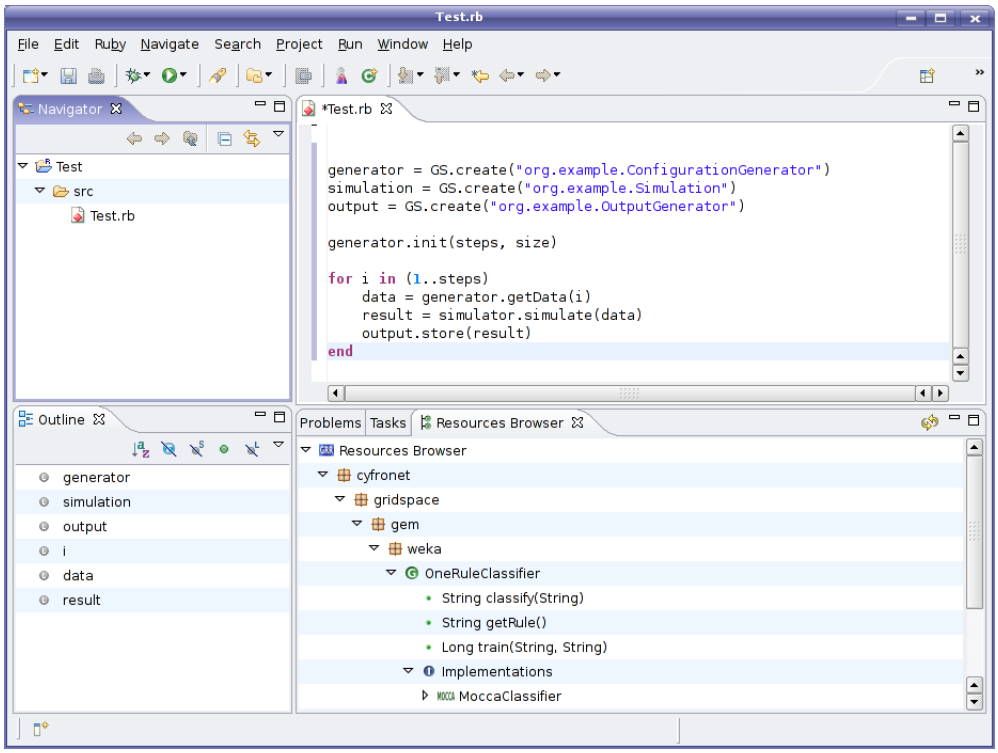

Figure 8: Eclipse Ruby Development Tools with example script and Registry browser

adapters for MOCCA and WS technologies, the support for current prototype of optimizer allows to specify optimization policy (goal) in a pluggable way, however no advanced algorithms have been implemented so far. For the monitoring system, the Gemini [61] monitoring infrastructure is being integrated.

Next step of development will consist in expanding the support for other component frameworks, and particularly implement allow the scripting language to instantiate GCM components.

4.5 Summary and Future Work

In this section we presented the concept of a high-level scripting language for programming component applications on the Grid. By using a dynamic interpreted language approach, it is possible to design a flexible and powerful notation, which covers all aspects of deployment, hierarchical and workflow composition, parametrization and configuration of components. The scripting approach can be applied to the process of rapid application development, prototyping and conducting scientific experiments on the Grid. The prototype which was developed demonstrates the feasibility of the proposed solution.

The future work includes enriching the programming language with a set of constructs for e.g. parallel execution, further development of deployment automation (optimization algorithms). Another interesting prospect which becomes open is the possibility to deploy developed scripts *as new components*. Such components could be then reused in more complex applications, as it is suggested by the GridSpace [38] concept.

5 Support for Master-worker Paradigm in GCM

This section defines a new construct for the ADL description of the GCM components allowing some abstract components to be specified. This extension to the ADL of the GCM is very general and allow the generic specification of a large number of composition patterns. This new construction allow the separation of a composition pattern, like e.g. a farm of components, from its implementation relying on real (non-abstract) components.

5.1 Motivation

The master-worker paradigm is very common in grid applications. However, with current component models, an application designer has to specify the number of workers as well as the mechanism to transport the requests from the master to the workers. However, these are complex issues. The number of workers depends on the available resources. Therefore, it should be handled by some adaptability systems. The mechanism to transport resquests from master to workers is also a complex issue. There is a lot of reasearch in this area around efficient scheduling algorithms and scalable architectures. For example, Network Enable Server (NES) such as DIET [18], Netsolve [19], or Ninf [58] implement advanced algorithms to find the best location where to handle a request while Desktop Grid systems such as SETI@Home [42], BOINC [65] or XtremWeb [17] target very large scale distributed systems.

The objective of this GCM extension is to relieve application designers for dealing with two difficult and low-level aspects: the number of worker and the request transport mechanism. Moreover, it aims at supporting legacy request transport middleware.

5.2 Overview of the model

The proposal to improve the support of the master-worker paradigm in GCM is based on the concept of *collection*. A collection is defined as a set of *exposed* ports, bound to some internal component type ports. A collection behaves like a component: it can be connected to other components. However, such a composition is done in an *abstract* architecture description, which represents the user's view of the application. Ideally at deployment time, or when resources are known, a collection is turned into a concrete assembly, formed by some internal component instances and by an instance of a *request transport pattern*. A *pattern* represents an implementation of an algorithm that specifies how to transport requests from *master* to *worker* components. Its implementation should be done by experts and it may be based on software components. Request transport *patterns* can be defined independently of collections. Figure 9 presents an overview of the proposed model's concepts.

5.3 A Master-Worker GCM Extension

5.3.1 Collection

A collection is a special kind of composite whose content may not be fully specified when it is defined: a collection definition describes only its exposed interfaces (names and types), the components types of its members and the binding between the external interfaces and the internal interfaces of these components types. Hence, the difference in a collection definition from a classical composite is to embed components types and to establishe connection between types. A graphical representation

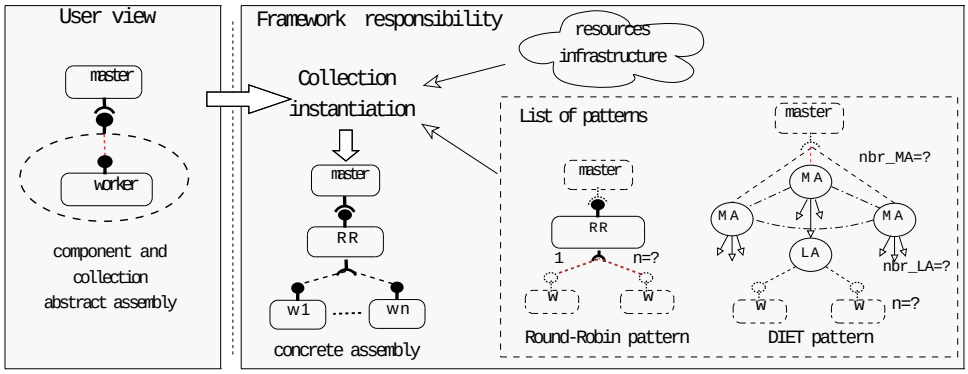

Figure 9: Overview of the master-worker GCM extension.

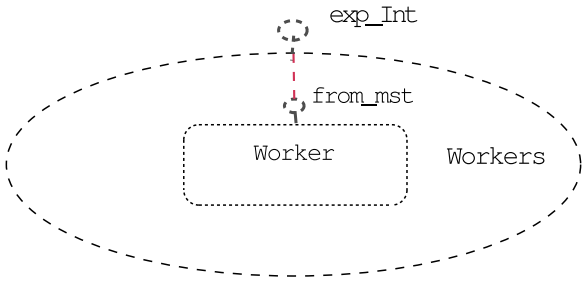

Figure 10: A collection description.

of a collection example is described in Figure 10: it defines a collection `Workers` with a server port `exp_Int` which is bound to a component type `Worker`.

As illustrated in Figure 11, a collection is a composite that contains one (or several) component types through the use of the `contentType` tag. The composite elements defined by this tag behave like regular components: their port can be connected to others composite or component ports. The only difference is that their actual number is left undefined. Optionally, an attribute can be added to the `contentType` tag to specify the number of instances. But using such an attribute should be reserved to very special situations.

5.3.2 An abstract ADL description

As a collection is not required to fully describe its contents, the resulting ADL description can not be directly instantiated. Such an ADL is said abstract.

The advantage of such an approach is that the conceptor of the application has not to change its way for building the application architecture. He/She has not to take care of the number of worker instances neither consider a possible request transport mechanism to be introduced between masters and the workers. They depend on the deployment environment and they need to be fully hidden to the conceptor. Then, he/she has not to be expert in master-worker environments like DIET, NetSolve, P2P, etc. to be able to design its application.

5.3.3 Request transport mechanism patterns

A collection need to be associated to a pattern to be turned into a concrete component that can be deployed. A pattern is a prototype or a skeleton of a request transport mechanism. It represents a technology used to transport a request from

```
<!--  components -->
<definition name="Worker">
  <interface name="from_mst" signature="Computation" role="server"/>
</definition>
<definition name="Worker_impl" extends="Worker">
  <content class="WorkerImpl"/>
</definition>

<!--  A collection definition -->
<definition name="Workers_impl" extends="Worker">
  <interface name="exp_Int" signature="Computation" role="server" />
  <contentType name="w" definition="Worker_impl"/>
  <binding client="this.exp_Int" server="w.from_mst"/>
</definition>
```

Figure 11: A collection definition.

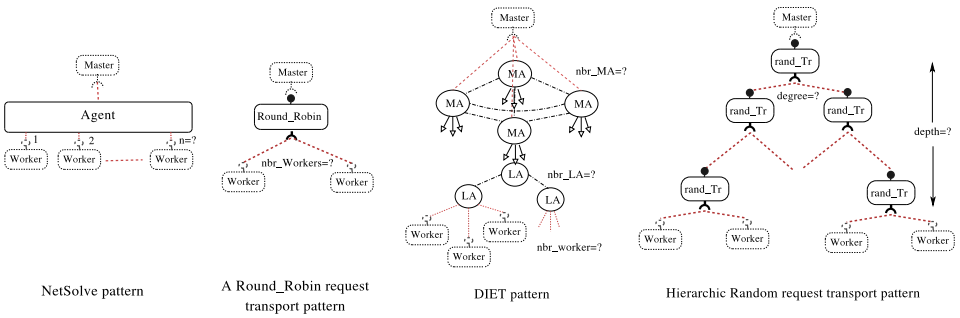

Figure 12: Different patterns example.

a master to worker components. It describes the components involved in this transport and their organisation: how they must be connected together and how the Master is bound to the Worker components.

A pattern can contain some undetermined variables. The role of these variables is to specify the number of the components in the pattern to be instanciated between a master and worker components. These variables essentially depends on the number of worker components to be instanciated. Therefore, such required information is an input variable of a pattern.

Figure 12 exposes some pattern examples. The dashed component instances represented in a pattern are extern elements expected to be linked to this pattern. The pattern representing NetSolve technology for instance, is composed of only one component: the *Agent*. The single information needed to instantiate this pattern is the number of connections to the worker components. This information is represented by the variable n. It is equal to the number of worker components to be instantiated.

More complex patterns are also exposed in Figure 12 like, for example, the DIET or the tree Random patterns. Because they are structured in a tree hierarchy, they are classified as hierachic patterns. The tree Random pattern for instance, is composed of multiple *rand_Tr* components structured in a tree architecture. The degree and the depth of this tree are represented respectivelly by the variables *degree* and *depth*. They depends on the number of the Worker components and a function can be associated to determine them. For example $depth = log(number\ of\ workers)$ and the *degree* value can be deduced from the resulted *depth*'s value.

```
 <!- A pattern definition ->
<definitionPattern name="RoundRobin">
  <transformation type="xsl" source="proxyRoundRobinGcm.xsl"/>
</definitionPattern>
<!--  A collection definition with a pattern tag -->
<definition name="Workers_impl" extends="Worker">
  <interface name="exp_Int" signature="Computation" role="server" />
  <contentType name="w" definition="Worker_impl"/>
  <binding client="this.exp_Int" server="w.from_mst"/>
  <pattern interface="this.exp_Int" transformationDef="RoundRobin"/>
</definition>
```

Figure 13: A collection definition which is associated to a pattern.

A pattern can be associated to a collection statically or dynamically. It can be associated statically by adding a `pattern` tag to a collection definition as illustrated in Figure 13. The pattern is associated to a port: distinct patterns can be used for distinct ports of a component. To dynamically associate a pattern to a collection, some collection oriented controllers are needed. They are out of the scope of this document.

Note that it is possible to associate several patterns to a port. Such patterns represents the list of allowed patterns that can be dynamically selected.

5.3.4 Pattern's selection and transformation step

The next step consists in selecting and applying an appropriate pattern to a collection. The selection depends essentially on the deployment environment. However, different deployment environements are possible to the same application. Hence, either the deployment tool or more preferably the adaptability framework are responsible for making choices. As it is out of the scope of this section to determine how such a choice is made, it is assumed that an oracle takes decisions. Figure 14 gives a summary of the pattern selection step. The patterns are those exposed in Figure 12.

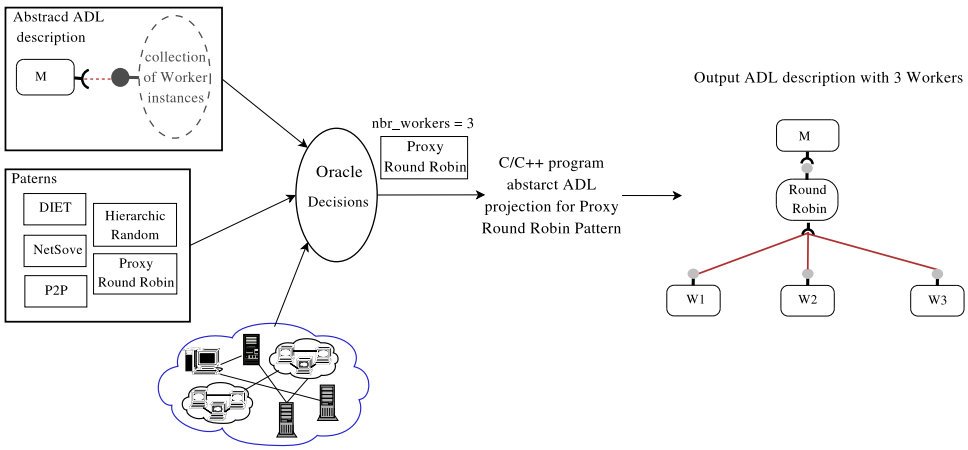

Figure 14: An example of a collection transformation.

After a pattern is selected, the application architecture can be accuratly specified. This step consists in projecting an abstract ADL description into a concrete

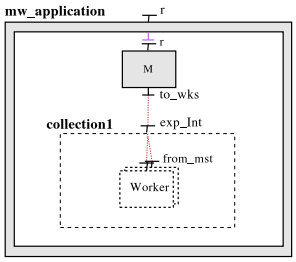

Figure 15: Example of a asbtract master-worker application.

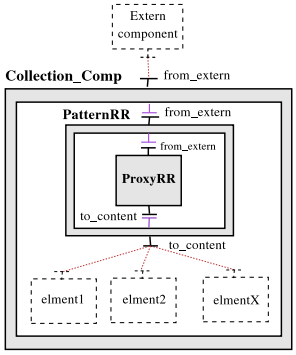

Figure 16: A Proxy Round-Robin pattern.

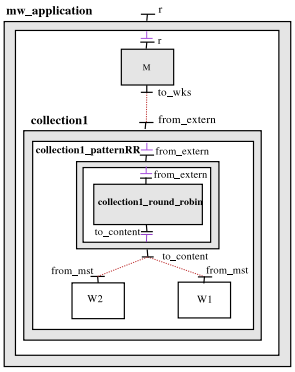

Figure 17: Concrete ADL obtained by applying the Round-Robin pattern.

ADL description. The input data are both the abstract description and the input paremeters of the selected pattern. A transformation results in a concret ADL description. This description contains the whole application architecture: the master instances, the worker instances, the instanciated components inside the selected pattern and connections between all these components. Thus, it can be deployed.

Figure 15 represents a collection definition while Figure 16 illustratres a round robin pattern. The result of applying such a round robin pattern to the previous collection with the number of workers set to 2 is represented in Figure 17.

Transformation implementation depends on the deployment tool, the controllers and/or the capabilities of the adaptability framework. They can a proiri be implemented in any language.

6 Optimizing Group Communication for GCM

This section shows how optimized group communication can be implemented in a component model, like e.g. the GCM. This section can be viewed as an efficient way to implement some of the collective interfaces defined in the GCM

GCM components that can be composed from several entities hosted on multiple distributed servers require an efficient communication methodology. In the following, we present the implementation of grid-enabled group communication procedures. As shown in [12], collective communications, like for example group communications are a central feature of a component model for the Grid, like the GCM.

The two shown procedures, a parallel broadcast and a parallel scatter implementation, have been developed within the scope of the HOC-SA project [28] for efficient communication among Higher-Order Components (HOCs [36]). Higher-Order Components were developed before the CoreGRID GCM was specified, independently of the CoreGRID project. Anyway, the features of HOCs (interoperability among heterogeneous servers, exchange of data and code over the network) are almost the same as the component features described in the GCM specification. Thus, any HOC application is also a candidate for a GCM implementation. The HOC broadcast and HOC scatter procedures are available as open-source software. Both are based on Java and were developed purely for efficient network communication, i. e. ,

these procedures are independent from the remaining HOC-SA tools and can also be used in the context of GCM (or other Java-based components).

We start the presentation of our optimized group communication procedures by introducing the internal parallel schema that leads to the high efficiency of the implementation. As a case study, we use the Alignment HOC: a component for DNA sequence processing, which requires to exchange large amounts of data over the network. The Alignment HOC takes as its parameters the DNA sequences to be processed and the application-specific part of the processing code. It is, therefore, a Higher-Order Component and makes use of the corresponding GCM feature (mobile code, *see* GCM specification (D.PM.04), Chapter 8). Finally, we compare the execution times of the collective communication procedures in the Alignment HOC with the corresponding ones in the popular ProActive library. ProActive [53] is a Java Grid middleware for parallel, distributed and multi-threaded computing which follows an active object pattern. Among some other features, it provides an implementation of the Fractal specification with some extensions, thus contributing to the development of the GCM. Note that ProActive was recently enhanced by *recursive doubling*, a feature similar to the presented technique. In our comparison, the ProActive version without *recursive doubling* was used.

Group communication in the HOC-SA

To share calculation data in the grid, the HOC-SA [28] provides two efficient group communication procedures for distributed networked computers: a broadcast and a scatter operation. Because programs running in the grid usually work on a large number of input elements and perform many compute-intensive operations on them, there is a strong motivation to distribute the data and share the processing load among multiple hosts.

The group communication procedures for data distribution in the HOC-SA are based on *orthogonal communication patterns* [54], which have been proven to be an efficient method of implementing MPI-based collective operations on local clusters. We implemented the orthogonal patterns using Java and RMI, allowing to communicate efficiently on a grid platform comprising distributed hosts of heterogeneous architectures.

Figure 18: Different group communication structures

Figure 18 shows two examples of group communication for eight grid nodes (i. e. , networked computers). On the left, it shows a linear group communication starting from node 'S0' to the nodes 'S1-S7', which leads to a bottleneck on the S0-link. Using our HOC group communications we avoid the bottleneck effect and speed up the whole operation. The communication structure used by our operations is created by continuously dividing the available nodes into a hierarchy of groups and subgroups until the deepest subgroups contain only two or less nodes. These groups can be graphically arranged in rows and columns, therefore, the name orthogonal communication.

The right part of Figure 18 shows the communication paths used in the HOC-broadcast in an example with eight nodes. The horizontal line represents the division in a top and a bottom group, each containing four nodes. Both groups contain two subgroups represented by dashed boxes.

During a group communication starting from the node S0, the message is first passed from the top group to one node in the bottom group (in the example in Fig. 18 from S0 to S4). Inside each of these groups, the message is then sent to the subgroups (from S0 to S2 and from S4 to S6). These two send operations are executed simultaneously. In the third and final step, the message is shared inside each of the four subgroups in four separate communication steps executed at the same time. Compared with the linear group communication in this example, the HOC operation needs only up to 3 sequential sending processes on each node, while the linear algorithm requires 7 sequential sending processes. Already for 16 grid nodes the theoretical speedup is about 73 % (15 sequential sending processes in the linear algorithm, four in the HOC group communication structure).

Currently, there is only a specification of GCM available but no full implementation, even if the GridCOMP project is on the process of implementing a reference implementation for the GCM over ProActive. Projects like HOCs or ProActive implement certain features of GCM (HOCs, e. g. , provide code mobility and ProActive provides components for hierarchical composition). Both, HOCs and ProActive offer group communication operations [11]. Contrary to the HOC implementation, ProActive implements group communication following a linear structure which leads to decreasing performance with a growing number of communicated data and participating grid nodes. To show the benefits of the HOC group operations, we compare the HOC-Broadcast with the corresponding group communication operation in ProActive.

To better understand the experiments, it should be noted that in ProActive, a broadcast or scatter operation does not only distribute data, but the data can be immediately processed: Java reflection is used to execute group operations on multiple hosts while supplying their input either in broadcast or scatter mode [11]. This mechanism provides a convenient abstraction over network communication, helping the ProActive user to concentrate on the application-level operations instead of data distribution, but it leads to a certain overhead. The HOC communication operations separate the data distribution from the data processing and are, therefore, at a lower level of abstraction than ProActive. But the HOC communication operations are only visible inside the component code (e. g. , the Alignment HOC) and never to the HOC user. When, e. g. , the Alignment HOC is used for genome processing, it encapsulates all the network communication, such that users who run a distributed sequence alignment benefit from the same kind of abstraction as ProActive users, with increased performance at the same time. To estimate the overhead of using reflection for running arbitrary user-defined methods on group-wise communicated data, we experimentally implemented another set of communication that follow a linear structure (like ProActive) but do not use reflection. To avoid I/O-delays, we use a thread pool and start multiple linear sending processes at once. Therefore, our experimental linear operations are called 'Multithreaded' in the following diagrams. The Multithreaded operations allow us to assess the pure communication costs of linear operations and compare them to ProActive and to our HOC communication operations.

Figure 19 shows the results for three broadcast operations. The left part of Figure 19 shows a test involving 16 grid nodes and a growing amount of data. The right part of the figure shows the same group communication, but with a fixed amount of data (25 MB) and a growing number of involved grid nodes. In both cases the advantage of our new HOC broadcast can be directly recognized from the diagram. Instead of exponentially growing communication times we achieved linear growth

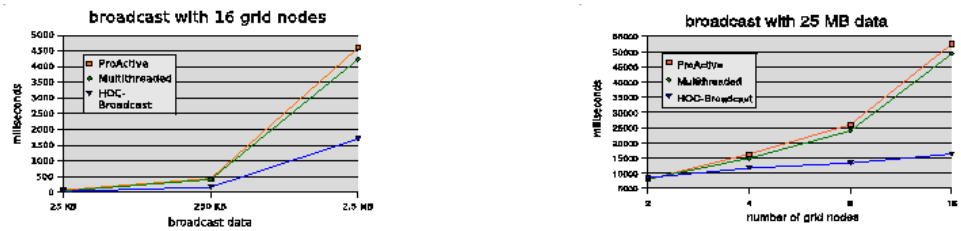

Figure 19: Results of the HOC broadcast experiments

when increasing the number of involved grid nodes. Thus, a parallel communication schema is strongly advisable for any Grid application. For GCM applications, it may also make sense compare the performance of the HOC operations with the new enhanced *recursive doubling* operations in ProActive (which has not been done, so far) before picking one implementation.

Our Alignment HOC is able to handle the pairwise processing of hundreds of megabytes of data (as present in total genome databases) by distributing the computations. The calculation power offered by the Alignment HOC makes it possible to keep up with the exponentially growing size of biological sequence databases when performing similarity detection and other kinds of biological data analysis applications.

7 The Integrated Toolkit: Supporting Grid Applications

7.1 Introduction

The Integrated Toolkit has been specified and designed in the framework of task 7.3 of the CoreGRID Institute on Grid Systems, Tools and Environments has as main objective the specification and design of an *Integrated Toolkit*: a framework which enables the easy development of Grid-unaware applications (those to which the Grid is transparent but that are able to exploit its resources) [24].

The Integrated Toolkit is mainly formed by an *interface* and a *runtime*. The former should give support to different programming languages, graphical tools and portals, and should provide the application with a small set of API methods. The latter should have the following features:

- The Grid remains transparent to the application, since the Integrated Toolkit performs all necessary actions to make this possible. The user is only required to select the tasks to be executed on the Grid and to use the API methods (maximum 5-6).
- Performance optimization of the application by exploiting its inherent concurrency. The possible parallelism is checked at task level, automatically deciding which tasks can be run at every moment. Consequently, the best suitable applications for the Integrated Toolkit are those with large granularity tasks.
- Task scheduling and resource selection taking into account task requirements and performance issues.

As a part of the STE generic component platform, depicted in Figure 20, the Integrated Toolkit builds on top of the Mediator Components and the Service and Resource Abstraction Layer.

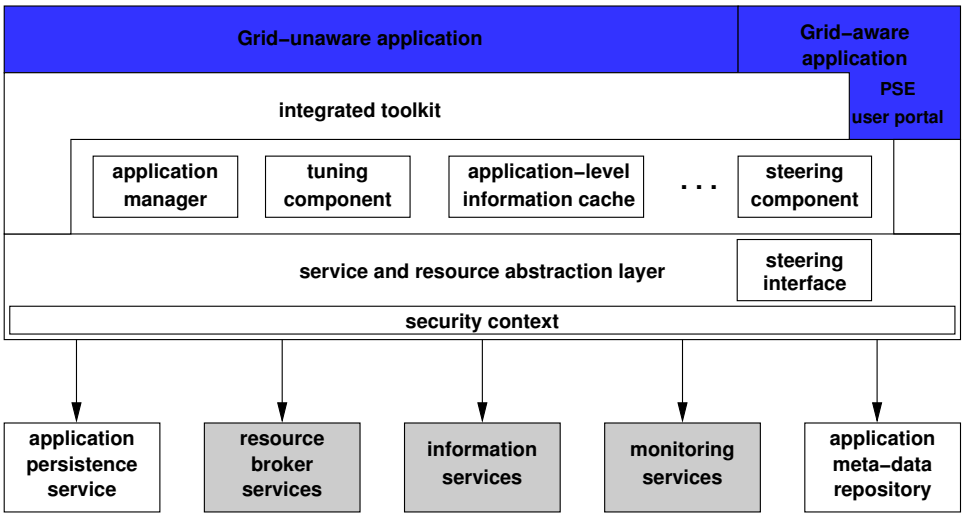

Figure 20: Envisaged generic component platform

The Mediator Components will offer system-component capabilities to the Integrated Toolkit, achieving both steering and performance adaptation. This set of components will allow to integrate Grid resources and services into one overall system with homogeneous component interfaces, while heterogeneous software architectures and technologies are situated underneath.

Furthermore, the Integrated Toolkit will also rely on a Service and Resource Abstraction Layer which represents an abstraction from the underlying Grid middleware, by means of a uniform interface for job submission, file transfer, resource management, ...

The Integrated Toolkit is based on the Grid Component Model (GCM). The design, inspired on the GRID superscalar framework [10], comprises the following components:

- *Task Analyser*: receives incoming tasks and detects their precedence, building a *task dependency graph*. It implements the Integrated Toolkit interface used by the application to submit tasks: when such a request arrives, it looks for data dependencies between the new task and all previous ones. When a task has all its dependencies solved, the Task Analyser sends it to the Task Scheduler.

- *Task Scheduler*: decides where to execute the dependency-free tasks received from the Task Analyser. This decision is made accordingly to a certain *scheduling algorithm* and taking into account three information sources: first, the available Grid resources and their capabilities; second, a set of user-defined constraints for the task; and third, the location of the data required by the task. This component could change its scheduling strategy on demand, thanks to the dynamic and reconfigurable features of the GCM.

- *Job Manager*: in charge of *job submission and monitoring*. It receives the scheduled tasks from the Task Scheduler and delegates the necessary file transfers to the File Manager. When the transfers for a task are completed, it transforms the task into a Grid job in order to submit it for execution on the Grid, and then controls the proper completion of the job. It could implement some fault-tolerance mechanisms in response to a job failure.

- *File Manager*: takes care of all the operations where files are involved, being able to work with both logical and physical files. It is a composite component which encompasses the File Information Provider and the File Transfer Manager components. The former gathers all information related with files: what kind of file accesses have been done, which versions of each file exist and where they are located. The latter is the component that actually transfers the files from one host to another; it also informs the File Information Provider about the new location of files.

In addition to providing support to Grid-unaware applications, we believe that our Integrated Toolkit design could also offer an alternative to develop Grid-aware applications. The componentised structure of the Integrated Toolkit makes possible to use it as a whole or to deploy solely specific subcomponents. For instance, a programmer interested in adding a scheduling functionality to an application could choose to deploy only the Task Scheduler subcomponent, binding its interfaces to the ones of the application components.

7.2 GCM features

The design of the Integrated Toolkit presented in Section 7.1 has been implemented using the ProActive library. The Integrated Toolkit exploits several GCM features which ProActive offers, namely the ones explained in next subsections.

7.2.1 Hierarchical composition

As said in Section 7.1, the Integrated Toolkit is a composite component that encompasses some inner subcomponents, presenting two levels of hierarchy. Each one of this subcomponents is a software unit responsible from a given functionality, aiming to follow the separation of concerns requirement. For instance, inside the Integrated Toolkit there is the File Manager (also a composite component), which is in charge of all the operations related with files; moreover, if we go deeper into the hierarchy, we can find two primitive components, the File Information Provider and the File Transfer Manager, which are themselves specializations of the functionality intended for the File Manager.

7.2.2 Functional and Non-functional Interfaces

On the one hand, the Integrated Toolkit provides *functional* server interfaces for the application to start/stop it, request the execution of tasks and open files to work with them locally. In addition, it uses functional client interfaces to inform the application about certain events (e.g. errors).

On the other hand, in addition to the default controllers provided by ProActive (life-cycle, binding, name, etc.) it is envisaged that the behaviour of the Integrated Toolkit will be modified using several *Application Controllers* which interact with Mediator Components. These controllers could be a part of the Integrated Toolkit membrane and offer non-functional server interfaces. Their possible roles could be:

- *Steering*: to modify certain parameters which affect the behaviour of the Integrated Toolkit. The Steering Controller could, for example, change the scheduling algorithm used by the Task Scheduler to match tasks with resources, or to specify whether the Job Manager should implement a fault tolerance feature when a job is likely to have failed.

- *Persistence*: to manage a checkpointing and fault recovery mechanism. In a given moment, the Persistence Controller could order the Task Analyser to perform a snapshot of the current application, in other words, to store which

tasks have successfully finished. This information could be used later, in case of an application or system failure, to resume the execution from the snapshot.

- *Distribution*: to inform of changes in resource state. For instance, the Distribution Controller could tell the Task Scheduler that a specific resource is no longer available.
- *Component*: to change the overall component structure of the Integrated Toolkit (e.g. to replace a component by another one with a different implementation).

7.2.3 Synchronous and Asynchronous Communications

The subcomponents of the Integrated Toolkit use bindings between client and server interfaces to interact. These communications can be either synchronous or asynchronous.

On the one hand, *synchronous communications* take place when the first component waits for the result of its invocation on the second one. For example, the Task Scheduler calls the synchronous method "newJob" on the Job Manager to request the creation of a job and waits for it to return the identifier of the new job, which will be used in later job state notifications.

On the other hand, *asynchronous communications* occur when the method call returns immediately since it does not expect any result. For instance, the Task Analyser tells the Task Scheduler about newly dependency-free tasks to schedule with the asynchronous method "scheduleTasks".

7.2.4 Collective Interactions

The GCM proposes two main kinds of parallel interfaces: multicast and gathercast. In the case of the Integrated Toolkit, *multicast communications* are used to transform a single invocation on a server interface into a list of invocations that are forwarded to all the subcomponents.

In particular, a multicast interface is used to both initialize and perform a cleanup in the subcomponents (see Figure 21).

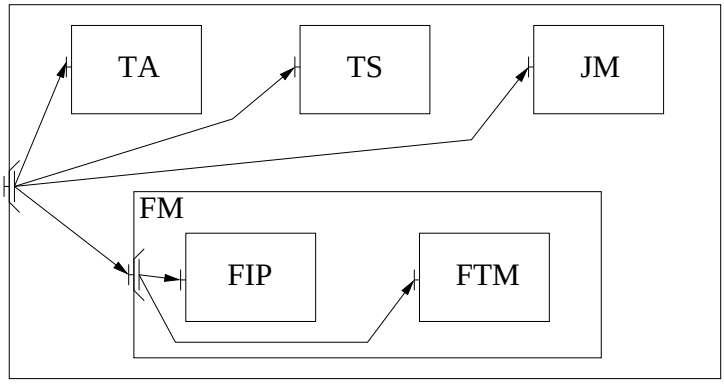

Figure 21: Multicast interface of the Integrated Toolkit

7.2.5 Deployment from ADL

In the case of Integrated Toolkit, the GCM component program architecture is described using a proper *ADL* (Architecture Description Language), that decouples

functional program development from the underlying tasks needed to deploy, run and control the components on the component framework.

In this sense, ProActive allows to load the definition of the Integrated Toolkit structure (components, interfaces, bindings) from ADL files, thus simplifying the instantiation process. This structure is mapped to a virtual node that is physically defined in a deployment descriptor.

7.3 Stopping a Structure of Components

During the execution of the application, the Integrated Toolkit runtime can experience errors of different kinds: a job submission that has failed, a problem with a file transfer, an exception in some point of the code, etc. Unfortunately, managing an error produced inside the Integrated Toolkit while it is working is not a trivial issue. Since all its subcomponents are interconnected and communicate constantly, a failure in one of them could impede the overall system to work properly.

The general response to such a situation should be to stop the components as quickly as possible; however, the components that form the Integrated Toolkit cannot be stopped in any arbitrary order because they have data dependencies. A dependency between two components A and B appears when A invokes a synchronous method on B and waits for its result. The problem arises if B is stopped before it can serve the request from A; in that case, A would remain blocked waiting for the result of the call and it could never serve the stop control request[1].

If one invokes the stop method of the Integrated Toolkit life-cycle controller (*stopFc*, see [50]) the call is forwarded to all the hierarchy of components in an *a priori* unknown order. Nevertheless, the synchronous calls between subcomponents lead to the dependencies shown in Figure 22, and such dependencies impose a stop order that must be respected; otherwise, we could experience a deadlock. One solution could be to redefine the Integrated Toolkit life-cycle controller to ensure that the subcomponents are stopped in an adequate order, specifically the following one: Task Analyser (TA), Task Scheduler (TS), Job Manager (JM), File Transfer Manager (FTM), File Information Provider (FIP).

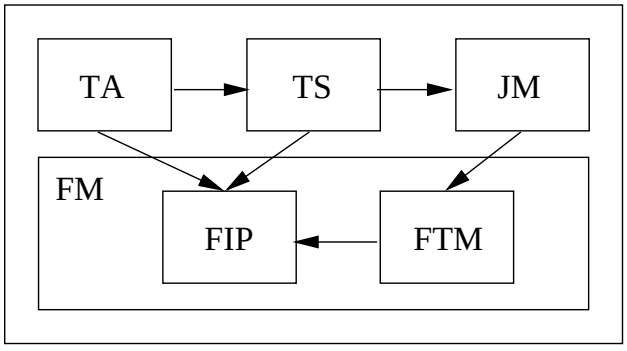

Figure 22: Data dependencies between Integrated Toolkit subcomponents

In conclusion, stopping a structure of components while they are serving requests can lead to a deadlock of the overall system if the dependencies that exist between them are not taken into account. Clearly, this is problem that should be managed by the implementations of the GCM.

[1]This theoretical behaviour has only been checked with ProActive components, therefore it might differ for other implementations of the Fractal/GCM model.

8 Autonomicity in the GCM: Autonomous Support of Grid Applications

This section details how the autonomicity feature described in the GCM specification can be implemented for some component patterns: the behavioural skeletons. In particular, this section details precisely an autonomic architecture for the farm pattern.

8.1 Advances in GCM Self-management Features

While developing grid applications neither the target platforms nor their status are fixed, statically or dynamically. This makes application adaptivity an essential feature in order to achieve high performance and to exploit efficiently the available resources. The basic use of static adaptation covers straightforward but popular methodologies, such as *copy-paste*, and *OO inheritance*. A more advanced usage covers the case in which adaptation happens at run-time. These systems enable dynamically defined adaptation by allowing adaptations, in the form of code, scripts or rules, to be added, removed or modified at run-time [14]. Among them is worth to distinguish the systems where all possible adaptation cases have been specified at compile time, but the conditions determining the actual adaptation at any point in time can be dynamically changed [8]. Dynamically adaptable systems rely on a clear separation of concerns between adaptation and application logic. This approach has recently gained increased impetus in the grid community, especially via its formalization in terms of the *Autonomic Computing* (AC) paradigm [49, 9, 5].

The CoreGrid Component Model (GCM) definition natively embodies the AC idea [23]. A GCM autonomic component consists of one or more managed components coupled with a single autonomic manager that controls them. To pursue its goal, the manager may trigger an adaptation of the managed components to react to a run-time change of application QoS requirements or to the platform status.

In this regard, an assembly of self-managed components implements, via their managers, a distributed algorithm that manages the entire application. Several existing programming frameworks aim to ease this task by providing a set of mechanisms to dynamically install reactive rules within autonomic managers. These rules are typically specified as a collection of `when-`*event*`-if-`*cond*`-then-`*act* clauses, where *event* is raised by the monitoring of component internal or external activity (e.g. the component server interface received a request, and the platform running a component exceeded a threshold load, respectively); *cond* is an expression over component internal attributes (e.g. component life-cycle status); *act* represents an adaptation action (e.g. create, destroy a component, wire, unwire components, notify events to another component's manager). Several programming frameworks implement variants of this general idea, including ASSIST [63, 3], AutoMate [51], SAFRAN [25], and finally the CoreGrid Component Model (GCM) [23]. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component model [50]. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids.

Though such programming frameworks considerably ease the development of an autonomic application for the grid (to various degrees), they rely fully on the application programmer's expertize for the set-up of the management code, which can be quite difficult to write since it may involve the management of black-box components, and, notably, is tailored for the particular component or assembly of them. As a result, the introduction of dynamic adaptivity and self-management might enable the management of grid dynamism, and uncertainty aspects but, at the same time, decreases the component reuse potential since it further specializes

components with application specific management code.

Within CoreGrid NoE and GridCOMP STREP, the *behavioural skeletons* concept has been proposed as a novel way to describe autonomic components in the GCM framework [1, 2]. Behavioural skeletons aim to describe recurring patterns of component assemblies that can be (either statically or dynamically) equipped with correct and effective management strategies with respect to a given management goal. Behavioural skeletons help the application designer to 1) design component assemblies that can be effectively reused, and 2) cope with management complexity by providing the programmer with component templates that, once instantiated, can be take part of general application management strategy spanning component assemblies in the horizontal (i.e. wiring) and the vertical (i.e. nesting) extent.

8.1.1 Describing Adaptive Applications

The architecture of a component-based application is usually described via an ADL (Architecture Description Language) text, which enumerates the components and describes their relationships via the *used-by* relationship. In a hierarchical component model, such as the GCM, the ADL describes also the *implemented-by* relationship, which represents the component nesting.

However, the ADL supplies a static vision of an application, which is not fully satisfactory for an application exhibiting autonomic behaviour since it may autonomously change behaviour during its execution. Such change may be of several types:

- *Component lifecycle.* Components can be started or stopped.
- *Component relationships.* The used-by and/or implemented-by relationships among components are changed. This may involve component creation, destruction, and component wiring alteration.
- *Component attributes.* A refinement of the behaviour of some components (which does not involve structural changes) is required, usually over a predetermined parametric functionality.

In the most general case, an autonomic application may evolve along adaption steps that involve one or more changes belonging to these three classes. In this regard, the ADL just represents a snapshot of the launch time configuration.

The evolution of a component is driven by its *Autonomic Manager (AM)*, which may request management action with the AM at the next level up in order to deal with management issues it cannot solve locally. Overall, it is a part of a distributed system that cooperatively manages the entire application.

An *Autonomic Behavioural Controller (ABC)* manages reconfiguration and monitoring of the component it belongs to. The ABC should be present in every component, even the passive ones in order to enact and monitor every part of an autonomic system.

In the general case, the management code executing in the AM of a component depends both on the component's functional behaviour and the goal of the management. The AM should also be able to cooperate with other AMs, which are unknown at design time due to the nature of component-based design. Currently, programming frameworks supporting the AC paradigm (such as the ones mentioned in Sec. 8.1) just provide mechanisms to implement management code. This approach has several disadvantages, especially when applied to a hierarchical component model:

- The management code is difficult to develop and to test since the context in which it should work may be unknown.

- The management code is tailored to the particular instance of the managed elements (inner components), further restricting the possible component reusability.

For this reason, an "ad-hoc" approach to management code is unfit to be a cornerstone of the GCM component model.

8.1.2 Behavioural Skeletons

Behavioural skeletons aim to abstract parametric paradigms of GCM component assembly, each of them specialized to solve one or more management goals belonging to the classical AC classes, i.e. configuration, optimization, healing and protection.

Behavioural skeletons represent a specialization of algorithmic skeleton concept for component management [21]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. Behavioural skeletons, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components.

Due to the hierarchical nature of GCM, behavioural skeletons can be identified with a composite component with no loss of generality (identifying skeletons as particular higher-order components [35]). Since component composition is defined independently from behavioural skeletons, they do not represent the exclusive means of expressing applications, but can be freely mixed with non-skeletal components. In this setting, a behavioural skeleton is a composite component that

- exposes a description of its functional behaviour;
- establishes a parametric orchestration schema of inner components;
- may carry constraints that inner components are required to comply with;
- may carry a number of pre-defined plans aiming to cope with a given self-management goal.

Behavioural skeleton usage helps designers in two main ways: the application designer benefits from a library of skeletons, each of them carrying several pre-defined, efficient self-management strategies; and, the component/application designer is provided with a framework that helps the design of new skeletons and their implementations.

The former task is achieved because (1) skeletons exhibit an explicit higher-order functional semantics, which delimits the skeleton usage and definition domain; and (2) skeletons describe parametric interaction patterns and can be designed in such a way that parameters affect non-functional behaviour but are invariant for functional behaviour.

8.1.3 A Basic Set of Behavioural Skeletons

Here a basic set of behavioural skeletons are presented for the sake of exemplification. Despite their simplicity, they cover a significant set of parallel computations of common usage.

One class of behavioural skeletons springs from the idea of *functional replication*. Let us assume the skeletons in this class have two functional interfaces: a one-to-many stream server S, and a many-to-one client stream interface C (see Fig. 23). The skeleton accepts requests on the server interface; and dispatches them to a number of instances of an inner component W, which may propagate results outside the skeleton via C interface.

Assume that replicas of W can safely lose the internal state between different calls. For example, the component has just a transient internal state and/or stores persistent data via an external data-base component.

Farm A stream of tasks is absorbed by a *unicast* S, each task is computed by one instance of W and sent to G, which collect tasks *from-any*. This skeleton can be equipped with a self-optimizing policy because the number of Ws can be dynamically changed in a sound way since they are stateless. The typical QoS goal is to keep a given limit (possibly dynamically changing) of served requests in a time frame. The AM just checks the average time tasks need to traverse the skeleton, and eventually reacts by creating/destroying instances of Ws, and wiring/unwiring them to/from the interfaces.

Data-Parallel A stream of tasks is absorbed by a *scatter* S; each task is split in (possibly overlapping) partitions, which are distributed to replicas of W to be computed. Results are *gathered* and assembled by G in a single item. As in the previous case, the number of Ws can be dynamically changed (between different requests) in a sound way since they are stateless. As in the previous case, the skeleton can be equipped with a self-configuration goal, i.e. resource balancing and tuning (e.g. disk space, load, memory usage), that can be achieved by changing the partition-worker mapping in S (and C, accordingly).

Active-Replication A stream of tasks is absorbed by a *broadcast* S, which sends identical copies to the Ws. Results are sent to G, which reduces them. This paradigm can be equipped with a self-healing policy because it can deal with Ws that do not answer, produce an approximate or wrong answer by means of a result reduction function (e.g. by means of averaging or voting on results).

The presented behavioural skeletons can be easily adapted to the case that S is a RPC interface. In this case, the C interface can be either a RPC interface or missing. Also, the functional replication idea can be extended to the stateful case by requiring the inner components Ws to expose suitable methods to serialize, read and write the internal state. A suitable manipulation of the serialized state enables the reconfiguration of workers (also in the data-parallel scenario [3]).

Anyway, in order to achieve self-healing goals some additional requirements on the GCM implementation level should be enforced. They are related to the implementation of the GCM mechanisms, such as the messaging system, the component membranes, and their parts (e.g. interfaces). At the level of interest, they are primitive mechanisms, in which correctness and robustness should be enforced ex-ante, at least to achieve some of the described management policies.

The process of identification of other skeletons may benefit from the work done within the software engineering community, which identified some common adaptation paradigms, such as *proxies* [56], which may be interposed between interacting components to change their interaction relationships; and dynamic *wrappers* [62]. Both of these can be used for self-protection purposes. As an example a couple of encrypting proxies can be used to secure a communication between components. Wrapping can be used to hide one or more interfaces whether a component is deployed into an untrusted platform.

8.2 Specifying Skeleton Behaviour

Autonomic management requires that, during execution of a system, components of the system are replaced by other components, typically having the same functional

behaviour but exhibiting different non-functional characteristics.

The application programmer must be confident about the behaviour of the replacements with respect to the original. The behavioural skeleton approach proposed supports these requirements in two key ways:

1. The use of skeletons with its inherent parametrization permits relatively easy parameter-driven variation of non-functional behaviour while maintaining functional equivalence.

2. The use of a formal or semi-formal specification to describe component behaviour gives the developer a firm basis on which to compare the properties of alternative realisations in the context of autonomic replacement.

The skeleton designer can use the description to prove rigorously (manually, at present) that a given management strategy will have predictable or no impact on functional behaviour. The quantitative description of QoS values of a component with respect to a goal, the automatic validation of management plans w.r.t. a given functional behaviour are interesting related topics, which are the subject of ongoing research. Examples of semi-formal specifications of the proposed skeletons can be found in [1, 4].

As byproduct, behavioural skeletons categorize GCM designers and programmers in three classes. They are, in increasing degree of expertize and decreasing cardinality:

- GCM users. They are supposed to use behavioural skeletons together with their pre-defined AM strategy. In many cases they should just instantiate a skeleton with inner components, and get as result a composite component exhibiting one or more self-management behaviour.

- GCM expert users. They are supposed to use behavioural skeletons overriding the AM management strategy. the personalization does not involve the ABC, thus does not need specific knowledge about GCM membrane implementation.

- GCM skeleton designers. They are supposed to introduce new behavioural skeletons or classes of them. At this end, the design and development of a brand new ABC might be required. This may involve the definition of new interfaces for the ABC, the implementation of the ABC itself together with its wiring with other controllers, and the design and wiring of new interceptors. This require a quite deep knowledge of the particular GCM implementation.

8.3 GCM Specification and Behavioural Skeletons

In terms of the GCM specification [23], a behavioural skeleton is a particular composite component exhibiting an autonomic conformance level strictly greater than one, i.e. a component with passive or active autonomic control. The component exposes pre-defined functional and non-functional client and server interfaces according to the skeleton type; functional interfaces are usually collective and configurable. Since skeletons are fully-fledged GCM components, they can be wired and nested via standard GCM mechanisms. From the implementation viewpoint, a behavioural skeleton is a partially defined composite component, i.e. a component with placeholders, which may be used to instantiate the skeleton. As sketched in Fig. 23, there are three classes of placeholders:

1. The functional interfaces S and C that are GCM functional interfaces, which may be equipped with monitoring interceptors controllers (currently objects).

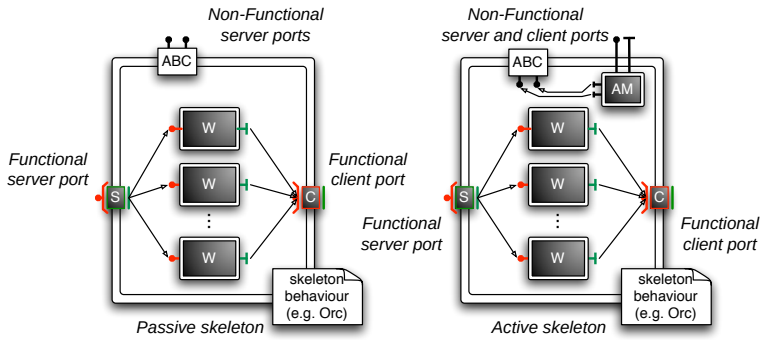

Figure 23: GCM implementation of functional replication. ABC = Autonomic Behaviour Controller, AM = Autonomic Manager, W = Worker component, S = Server interface (one-to-many communication e.g. broadcast, data-parallel scatter, unicast), C = Client interface (many-to-one communication e.g. from-any, data-parallel gather, reduce, select).

2. The AM that is a particular inner component. It includes the management plan, its goal, and exported non-functional interfaces.

3. Inner component W, implementing the functional behaviour.

The orchestration of the inner components, and thus ABC functionality, is implicitly defined by the skeleton class. In order to instantiate the skeleton, placeholders should be filled with suitable entities. Observe that just entities in the former two classes are skeleton specific.

In addition to a standard composite component, a behavioural skeleton is further characterized by a formal (or semi-formal) description of the component behaviour. This description can be attached to the ADL component definition via the standard GCM ADL hook, which can be used with any behavioural specification language. In this regard, a description based on the Orc language have been proposed within GridCOMP and CoreGrid projects [1, 4].

8.4 Extension of GCM collective communications

The GCM natively provides the programmer with collective interfaces, so-called *multicast* and *gathercast* (see [23]). These interfaces aim to split-and-distribute/gather-and-join, respectively, a single RPC call or stream item to/from multiple components. These interfaces, however, are not able to cope with typical operations performed on stream computation. In particular, they do not deal with many, possibly consecutive, stream items (or RPC calls). Within this deliverable, we introduce in the GCM the *unicast* collective interface, thus extending the set of GCM collective interfaces. This interface is part of a set of interfaces specifically designed to cope with the distribution of consecutive stream items from a single *source_interface* toward a *target_interface* dynamically chosen in set. The interfaces in the set typically belong to different components, while the *target_interface* is dynamically chosen at the dispatch time of each item knowing the history of previous choices. As an example, this enable to dispatch consecutive items in a stream (or consecutive calls of a method) toward different components in round-robin fashion. Previously mentioned *from-any* interface covers the collection of items in a similar fashion. Several variants of this kind of interfaces can be imagined, as an example *unicast-on-demand*, *from-any-ordered*, and *from-any-ordered*. Previous works with the ASSIST coordi-

nation language proved the expressiveness and efficiency of these kind of interfaces [63, 3].

8.5 Contribute Summary

The challenge of autonomicity in the context of component-based development of grid software is substantial. Building into components autonomic capability typically impairs their reusability. Behavioural skeletons have been proposed as a compromise: being skeletons they support reuse, while their parametrization allows the controlled adaptivity needed to achieve dynamic adjustment of QoS while preserving functionality. We have described how these concepts can be applied and implemented within the GCM. We have introduced a significant set of skeletons, together with their self-management strategies. We sketched the GCM implementation of a class of those (functional replication class), that have been exemplified via the farm skeleton. The presented behavioural skeletons have been implemented in GCM-ProActive [13], in the framework of the CoreGrid and GridCOMP project and are currently under extensive experimental evaluation. Preliminary results, confirm the feasibility of the approach.

9 Conclusion

This document illustrated the innovative aspects of the GCM by showing how it can be used in various research and applied areas, either for implementing Grid platforms, or for easing the design, definition and deployment of Grid applications, or for improving the performance of Grid applications.

Outside the contributions presented in this deliverable, GCM has also been used in different settings showing the effectiveness of the component model. First, a prototype of the ProActive implementation of the GCM has already been used to build and deploy over a Grid a numerical computation application for electromagnetism [52]. Moreover, in the context of the *common component modeling example (CoCoME)* context, GCM components have been modeled and specified, and a prototype implementation has been realized [16]. The CoCoME consists of a point-of-sale example featuring distribution, asynchronism, and collective communications.

Concerning interoperability between the GCM and other standards or component models; the CoreGrid community has demonstrated the capacities of the GCM in the past, first through effective interactions between CCA and GCM components [44], and second by the possibility to expose component interfaces as web services [27].

Next steps of development for the GCM include the experimentation of interactions between new GCM frameworks, and between GCM component and other frameworks. Experiments within the Integrated Toolkit are of particular interest to show the effectiveness of the GCM implementations, and the appropriateness of the GCM for developing Grid applications and Middlewares.

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Peter Kilpatrick, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons for component autonomic management on grids. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Creete, Greece, 2007.

[2] Marco Aldinucci, Sonia Campa, Patrizio Dazzi, and Nicola Tonellotto. *D.NFCF.01 – Non functional component subsystem architectural design*. GridCOMP STREP deliverable D.NFCF.01, June 2007.

[3] Marco Aldinucci and Marco Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006. DOI:10.1016/j.parco.2006.04.001.

[4] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Management in distributed systems: a semi-fomal approach. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Proc. of 13th Intl. Euro-Par 2007 Parallel Processing*, LNCS, Rennes, France, August 2007. Springer. To appear.

[5] Marco Aldinucci, Marco Danelutto, and Marco Vanneschi. Autonomic QoS in ASSIST grid-aware components. In Beniamino Di Martino and Salvatore Venticinque, editors, *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pages 221–230, Montbéliard, France, February 2006. IEEE.

[6] B. A. Allan et al. The CCA core specification in a distributed memory SPMD framework. *Concurrency Computat.*, 14:1–23, 2002.

[7] Ilkay Altintas, Efrat Jaeger, Kai Lin, Bertram Ludaescher, and Ashraf Memon. A web service composition and deployment framework for scientific workflows. *ICWS*, 0:814, 2004.

[8] F. André, J. Buisson, and J.-L. Pazat. Dynamic adaptation of parallel codes: toward self-adaptable components for the Grid. In *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer, January 2005.

[9] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt. On adaptability in grid systems. In *Future Generation Grids*, CoreGRID series. Springer, November 2005.

[10] Rosa Badia, Jesus Labarta, Raul Sirvent, Josep M. Perez, Jose M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[11] Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Java Grande Conference*, pages 28–36, Seattle, 2002. ACM Press.

[12] Francçoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *CCGrid 2007: IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610, May 2007.

[13] Françoise Baude, Denis Caromel, and Matthieu Morel. On hierarchical, parallel and distributed components for grid programming. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 97–108, Saint-Malo, France, January 2005. Springer.

[14] Jan Bosch. Superimposition: a component adaptation technique. *Information & Software Technology*, 41(5):257–273, 1999.

[15] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of Seventh International Workshop on Component-Oriented Programming*, June 2002.

[16] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu. *A Specification Language for Components Implemented in GCM/ProActive*. LNCS series. Springer Verlag, 2007. to appear.

[17] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, FrÌ©dÌ©ric Magniette, Vincent NÌ©ri, and Oleg Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *FGCS*, 21(3):417–437, 2005.

[18] E. Caron, F. Desprez, F. Lombard, J.M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[19] H. Casanova and J. Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[20] CORBA Component Model, v3.0, 2002. `http://www.omg.org/technology/documents/formal/components.htm`.

[21] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[22] Massimo Coppola, Marco Danelutto, Sbastien Lacour, Christian Prez, Thierry Priol, Nicola Tonellotto, and Corrado Zoccolo. Towards a common deployment model for grid systems. In Sergei Gorlatch and Marco Danelutto, editors, *Proc. of the Integrated Research in Grid Computing Workshop*, volume TR-05-22, pages 31–40, Pisa, Italy, November 2005. Universit di Pisa, Dipartimento di Informatica.

[23] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, February 2007.

[24] CoreGRID NoE deliverable series, Institute on System, Tools and Environments. *Deliverable D.STE.05 – Design Of The Integrated Toolkit With Supporting Mediator Components*, November 2006.

[25] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Proc of the 5th Intl Symposium Software on Composition (SC 2006)*, volume 4089 of *LNCS*, pages 82–97, Vienna, Austria, March 2006. Springer.

[26] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing: Second European Across-Grids Conference, AxGrids*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2004.

[27] J. Dünnweber, F. Baude, V. Legrand, N. Parlavantzas, and S. Gorlatch. *Invited papers from the 1st CoreGRID Integration Workshop, Pisa, Novembre 2005*, chapter Towards Automatic Creation of Web Services for Grid Component Composition. volume 4 of CoreGRID series. Springer, jan 2006. ISBN: 0-387-27935-0. Also as the CoreGrid TR003 report, `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0003.pdf`.

[28] Jan Dünnweber, Philipp Lüdeking, Cătălin L. Dumitrescu, Eduardo Argollo, and Sergei Gorlatch. The HOC-SA Globus Incubator Project. Web page: http://dev.globus.org/incubator/hoc-sa/, 2006.

[29] E.Bruneton, T.Coupaye, and J.B. Stefani. The Fractal Component Model `http://fractal.objectweb.org/specification/index.html`. Technical report, ObjectWeb Consortium, February 2004.

[30] Eclipse - an open development platform. `http://www.eclipse.org/`.

[31] Eclipse graphical editing framework. `http://www.eclipse.org/gef/`.

[32] D. Gannon and G. Fox. Workflow in grid systems meeting. *Concurrency & Computation: Practice & Experience*, 18(10), 2006. Based on GGF10 Berlin Meeting.

[33] Deliverable D.PM.02 - proposals for a Grid component model, 2006. http://www.coregrid.net.

[34] Vladimir Getov and Thilo Kielmann, editors. *Component Models and Systems for Grid Applications.* Springer, 2005.

[35] S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer, November 2005.

[36] Sergei Gorlatch and Jan Dnnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*, pages 299–306. Springer Verlag, 2005.

[37] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, and Randall Bramley. Merging the CCA component model with the OGSI framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 182, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Tomasz Gubala and Marian Bubak. Gridspace - semantic programming environment for the grid. In Wyrzykowski et al. [64], pages 172–179.

[39] Tomasz Gubala and Andreas Hoheisel. Highly dynamic workflow orchestration for scientific applications. In *CoreGRID Intergation Workshop 2006 (CIW06)*, pages 309–320. ACC CYFRONET AGH, 2006.

[40] Java powered Ruby implementation, 2007. `http://jruby.codehaus.org/`.

[41] The Jython Website, 2004. `http://www.jython.org`.

[42] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. In *IEEE Computer Society*, pages 78–83, Los Alamitos, CA, USA, February 2001.

[43] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proc. Int. Workshop on High-Level Parallel Progr. Models and Supportive Environments (HIPS)*, pages 90–97, Santa Fe, New Mexico, USA, April 2004.

[44] M. Malawski, M. Bubak, F. Baude, D. Caromel, L. Henrio, and M. Morel. Interoperability of grid component models: GCM and CCA case study. In *CoreGRID Symposium in conjunction with Euro-Par 2007*, CoreGRID series. Springer Verlag, august 2007.

[45] Maciej Malawski, Marian Bubak, Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Interoperability of grid component models: GCM and CCA case study. In *Proceedings of the CoreGRID Symposium*, Rennes, France, 2007. Springer. to appear.

[46] Maciej Malawski, Dawid Kurzyniec, and Vaidy Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005)*, 2005.

[47] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *UK e-Science All Hands Meeting*, pages 627–634, Nottingham, UK, September 2003. ISBN 1-904425-11-9.

[48] Anthony Mayer, Steven Newhouse, and John Darlington. A Software Architecture for HPC Grid Applications. In *6th International Euro-Par Conference*, pages 686–689, 2000.

[49] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, January 2006.

[50] ObjectWeb Consortium. *The Fractal Component Model, Technical Specification*, 2003.

[51] Manish Parashar, Hua Liu, Zhen Li, Vincent Matossian, Cristina Schmidt, Guangsen Zhang, and Salim Hariri. AutoMate: Enabling autonomic applications on the Grid. *Cluster Computing*, 9(2):161–174, 2006.

[52] N. Parlavantzas, M. Morel, V. Getov, F. Baude, and D. Caromel. Performance and scalability of a component-based grid application. In *9th Int. Worshop on Java for Parallel and Distributed Computing, in conjunction with the IEEE IPDPS conference*, April 2007.

[53] ProActive home page, 2006. `http://www-sop.inria.fr/oasis/proactive/`.

[54] T. Rauber, R. Reilein-Ruß, and G. Rünger. ORT - A Communication Library for Orthogonal Processor Groups. In *Proc. of the ACM/IEEE Supercomputing Conf. 2001 (SC'01)*, Denver, Colorado, USA, 2001. ACM.

[55] The Ruby programming language, 2007. `http://www.ruby-lang.org`.

[56] S. M. Sadjadi and P. K. McKinley. Transparent self-optimization in existing CORBA applications. In *Proc. of the 1st Intl. Conference on Autonomic Computing (ICAC'04)*, pages 88–95, Washington, DC, USA, 2004. IEEE.

[57] Raul Sirvent, Josep M. Perez, Rosa Badia, and Jesus Labarta. Automatic grid workflow based on imperative programming languages. *Concurrency and Computation: Practice and Experience*, 18:1169–1186, 2005.

[58] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *J. Grid Compututing*, 1(1):41–51, 2003.

[59] Bruce A. Tate. *Beyond Java*. O'Reilly, 2005.

[60] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual grid workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[61] Hong Linh Truong, Bartosz Balis, Marian Bubak, Jakub Dziwisz, Thomas Fahringer, and Andreas Hoheisel. Towards distributed monitoring and performance analysis services in the k-wfgrid project. In Wyrzykowski et al. [64], pages 156–163.

[62] E. Truyen, B. Jørgensen, W. Joosen, and P. Verbaeten. On interaction refinement in middleware. In J. Bosch, C. Szyperski, and W. Weck, editors, *Proc. of the 5th Intl. Workshop on Component-Oriented Programming*, pages 56–62, 2001.

[63] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

[64] Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors. *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, volume 3911 of *Lecture Notes in Computer Science*. Springer, 2006.

[65] Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu/`, 2002.