Proceedings of the
Programming Model Institute Technical meeting 2008

edited by
M. Danelutto
Dept. Computer Science – Univ. Pisa

CoreGRID Technical Report
Number TR-XXX
May 1, 2008

Institute on Programming Model
CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme
Project no. FP6-004265
Abstract

In January 2008, the Programming model Institute of CoreGRID had a plenary meeting in Paris. The meeting was aimed at discussing the progresses achieved on the Programming model Institute research themes. During the meeting – most of the Institute partners were attending – different researchers presented their on going work related to the Institute research themes. It’s worth pointing out this has been the first technical meeting were the new Associate partners of CoreGRID (the ones participating to the Programming model Institute activities) participated actively.

This report hosts a synthetic version of the work presented in the Paris meeting. The different contributions cover the most important research themes of the Institute. In particular, three contributions are related to autonomic features in GCM (the ones presented in Sec. 4, 6 and 7), three deal with workflow/component related themes (Sec. 3, 8 and 9). Overall, all the contributions are more or less directly related to GCM and, more in general, to component programming models.

---

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).
## Contents

1 Adapting Loop Parallelization to the Grid  
**by M. Classen, P. Classen, C. Lengauer (UNI-PASSAU), J. Duenneber, S. Gorlatch (WWU Muenster)**  
1.1 Contribution ................................................................. 3  
1.1.1 Largely Decentralized Communication Scheme .......................... 3  
1.1.2 The Controller ............................................................ 3  
1.1.3 The Clients ............................................................... 4  
1.2 Automatic Loop Parallelization ............................................. 4  
1.2.1 Integration of the LooPo-HOC with the Middleware .................... 4  
1.3 Results .......................................................... 5

2 Component Measurable Values and Services: a Technology for the Conclusion of Resource Transactions  
**by N. Currle-Linde (HRLS), C. Perez (INRIA), M. Coppola (ISTI/CNR)**  
2.1 Contribution ................................................................. 6  
2.1.1 GCM component description ............................................ 6  
2.1.2 Measurable property definition ....................................... 7  
2.1.3 Time Features of Application Programs. .............................. 8  
2.1.4 Interaction between Applications and the Grid ....................... 8  
2.2 Positioning in the Institute activities ................................. 9

3 Towards a spatio-temporal component model  
**by C. Perez, H. Bouziane (INRIA)**  
3.1 Contribution: A spatio-temporal model based on GCM and AGWL ............ 10  
3.1.1 Extending GCM components with tasks and temporal ports .......... 11  
3.1.2 Life cycle management of task-components .......................... 11  
3.1.3 A composition language based on a modified AGWL ................. 12  
3.2 Positioning in the Institute activities ................................. 13

4 A component platform for experimenting with autonomic composition  
**by L. Henrio, D. Caromel, F. Baude (INRIA)**  
4.1 Contribution ................................................................. 14  
4.1.1 Programming Autonomic Applications .................................. 14  
4.1.2 Structuring Non-functional Concerns with Components ............. 15  
4.1.3 Plugging Autonomic Behaviours Inside Components ................ 16  
4.1.4 Autonomic Distributed Services ..................................... 16  
4.2 Positioning in the Institute activities ................................. 17  
4.2.1 Extensions, Implementation, and Refinement of the GCM .......... 17  
4.2.2 A General Framework for Autonomic applications, based on the GCM 18  
4.2.3 A Loosely Coupled GCM Model ................................. 18

5 Deductive verification of GCM: Deontic Temporal Resolution  
**by A. Bolotov (UOW)**  
5.1 Contribution ................................................................. 19  
5.2 Positioning in the Institute activities ................................. 19

6 Experiments with GCM and SCA: preliminary experiences and results  
**by M. Danelutto, G. Zoppi (UNIPI)**  
6.1 Contribution ................................................................. 20  
6.1.1 Implementation .......................................................... 21  
6.1.2 Experimental results .................................................. 21  
6.2 Positioning in the Institute activities ................................. 22
7 Advances in GCM
by N. Tonellotto, P. Dazzi (ISTI/CNR),
M. Aldinucci, S. Campa, M. Danelutto (UNIPI) 24
7.1 Contribution ........................................ 24
7.2 Implementation .................................... 25
7.3 Positioning in the Institute activities .................. 27

8 Secure component composition and definition with QoS
by J. Cohen, S. Mc Gough (IC) 28
8.1 Information, information, information .................. 28
8.2 Workflow Validation and Optimisation .................. 29
8.3 Workflow Pipeline ................................... 30
8.4 Component Abstraction ................................ 30
8.5 Workflow Scheduling/Planning ......................... 31
8.6 Quality of Service (QoS) ............................. 32
8.7 Workflow Security ................................... 32
8.8 Related Projects .................................... 33

9 Workflows in ProActive/GCM
by N. Ranaldo, E. Zimeo (UoS) 35
9.1 Contribution .......................................... 36
9.1.1 ProActive/GCM and the ProActive Scheduler ....... 36
9.1.2 SAWE ............................................. 36
9.1.3 SAWE and ProActive-based Activities ............... 37
9.2 Positioning in the Institute activities .................. 38

10 WP3 activities at UoL
by J. Cunha (UoL) 39
10.1 Contribution .......................................... 39
10.2 Positioning in the Institute activities .................. 39
1 Adapting Loop Parallelization to the Grid

by M. Classen, P. Classen, C. Lengauer (UNI-PASSAU),
J. Duennweber, S. Gorlatch (WWU Muenster)

This work is an extension to our approach to simplifying the programming of Grid applications by combining 1) the automatic loop parallelization compiler LooPo which can resolve complex data and control flow dependences [7, 8] with 2) Higher-Order Components (HOCs [9]), i.e., components that are readily integrated with the target middleware [10]. In this approach, developers specify only the application-specific tasks of their programs for the Grid and upload these tasks as parameters to a specific HOC, called LooPo-HOC. In the case that data dependences exist between the tasks, the scalability of the generated code depends largely on the amount of communication caused by the applied communication scheme [11]. Therefore, we extend the LooPo-HOC by a largely decentralized communication scheme which leads to maximum scalability [12].

1.1 Contribution

1.1.1 Largely Decentralized Communication Scheme

Although P2P communication is always decentralized, in our scheme, a dedicated controller keeps track of the dependences between the tasks being processed. This controller keeps shared data up-to-date, while the bulk of the communication (among the clients) is still performed independently.

Figure 1 gives an overview about the interaction between the controller and several clients. The individual responsibilities of controller and client are as follows.

1.1.2 The Controller

The controller manages and schedules the tasks for execution and controls the communication. The tasks are stored in a task graph, wherein nodes represent tasks and edges dependences between tasks defining the execution order. The communication is controlled by inducing a connection between dependent tasks (placed on client PCs).

![Diagram](image)

Figure 1: Typical sequence
1.1.3 The Clients

In the employed largely decentralized communication pattern, the clients execute the tasks and communicate data to the dependent communication partner. The controller assigns new tasks to the clients via the scheduler (by sending them so-called soaking data, Fig. 2(a)), according to the task graph. Whenever other tasks depend on the result of a task, the client receives from the controller also the address of the destination computer whereto the result is sent (Fig. 2(b)). If a client executes a task that depends on another task, the required data is already buffered (Fig. 2(c)). It must only be unpacked, transferred to a send-buffer (for forwarding the result to the next client, Fig. 2(d)) and is then processed locally (Fig. 2(e)). At the end of the task execution, all final values of data elements are typically scattered across the clients. The action of collecting these values and returning them to the controller is called draining and is governed by the controller (Fig. 2(f)).

1.2 Automatic Loop Parallelization

In order to obtain a distributed program consisting of tasks that can be executed in parallel, we use methods for the automatic parallelization of program loops. These methods are based on a mathematical model, the Polytope Model.

We have adapted the classic tiling technique for dynamic task farming: the controller generates more tiles (groups of tasks) than physical processors are available in order to balance the workload among the farm workers.

1.2.1 Integration of the LooPo-HOC with the Middleware

A Web service is used for remote access to the LooPo HOC and the distributed application state (status data and intermediate results) is maintained via a resource configuration, as is typical in the Web Service Resource Framework (WSRF).

While the service interface itself is stateless, the resources connected to it (as configured in a setup file) hold their state (in the form of transient variables, called resource properties in WSRF) even past the scope/duration of a session. The LooPo-HOC makes use of this feature, e.g., for parallelizing a loop nest and preserving the resulting task graph as a data record in a resource, which can be referenced by a key and reused in multiple applications.

Another feature, through which the LooPo-HOC benefits from the WSRF middleware, is its support for asynchronous operations. While LooPo transforms loop nests, Web clients can disconnect or even shut down. The LooPo-HOC can restore the task graph from a former session, when a client sends it the corresponding resource key. The LooPo-HOC uses two types of WSRF resources. For every code transformation request, one new resource instance (i.e., transient storage) for holding the resulting task graph is created dynamically. The other resource is static (i.e., instantiated only once and shared globally among all processes), and used for workload monitoring.
The task graph resources are instantiated following the factory pattern, returning a unique remote reference (the resource key) to the client. As shown in Fig. 3, the client sends the resource key on every communication with the LooPo-HOC, which uses the key afterwards to retrieve the corresponding resource data (the task graph and intermediate results). Thus, a LooPo-HOC server is not a single point of failure, but rather a service provider that permits clients to switch between mirror hosts during a session.

1.3 Results

We tested our approach in experiments on the examples of matrix multiplication, successive over-relaxation (SOR) and polynomial product. Our experiments proved that our largely decentralized communication scheme leads to better scalability than standard task farming. In the example of the polynomial product algorithm, the program scaled for up to 32 processors. The examples of matrix multiplication and SOR, the problems sizes were not large enough to achieve scalability for more than 16 processors. However, our peer-to-peer implementation still resulted in speedup improvements as compared to a centralized task farm.

Further optimizations can be achieved in the distributed arrays with regard to the memory consumption of the clients.
2 Component Measurable Values and Services: a Technology for the Conclusion of Resource Transactions

by N. Currle-Linde (HRLS), C. Perez (INRIA), M. Coppola (ISTI/CNR)

The rapid development of Grid technologies [45] over the past few years has made it possible to solve a number of new problems. However, the wealth of new tools and approaches has made it necessary to evaluate these methods applied in Grid experiments. Of specific interest is the investigation of methods for the interaction between clients and owners of resources.

The Grid Component Model (GCM) [66] aims at mastering both the complexity of applications and resources. However, so far there has been little effort to try to automate the deployment of GCM applications on Grids. The GCM Architecture Description Language (ADL) is not intended to be easily edited by end users, as it contains information about the mapping of components onto resources. Such a mapping requires a high level of expertise on applications and on resources.

In order to support the efficient execution of complex scientific or engineering applications, an adequate organization of services for resource distribution is needed. The existing organization of the services available does not support reliable relations between individual clients and those who represent their interests, i.e. the Grid services, because these services are aimed at the simultaneous realization requirements of a multitude of clients and of the owners of the Grid resources. As there are no high-level instruments available for the automated planning of the execution of complex applications, it is generally not possible to provide services of an appropriate quality-level for these applications.

As service and/or resource failures are inherent properties of grids, there must be a control and a monitoring of applications and of the resources, so that the application to be executed can at every moment reach a maximum level of productivity. An example of such organization has been proposed in [46] in order to design a deployment model for Grids. However, that approach does not take into account dynamic properties that can be attached to a component.

This report deals with dynamic properties and their use in relating the execution of complex applications in Grid organizations to the establishment of traditional market relations between users (clients) and owners of Grid resources. In particular, we propose the notion of measurable components as a foundation technology. An example of the needed supporting services for the notion can be found in the distributed directory service in development as part of the XtreemOS research project [50].

2.1 Contribution

There are several situations that need to deal with the description of a component. The main example of such a situation is the deployment phase that selects a particular component implementation as well as the resource on which it will be deployed. Existing deployment tools mainly focus on a static description of a component, which takes into account properties like the architecture of the processor the implementation has been compiled to, the operating system, the implementation language and framework, etc. However, dynamic properties like memory consumption, wall-time, etc. are seldom taken into account, but they are often essential in order to perform a relevant choice of deployment. Such is the case when, in a market relation, the user imposes economic constraints on the resources to be selected, relating resource cost with resource reliability and performance.

We believe it is important to propose a general solution, enabling static and dynamic properties to be attached to a component, and imposing no a priori restrictions on the set of properties used to direct deployment and adaptivity of components.

2.1.1 GCM component description

Components are statically described by the set of ports they expose, possibly by their implementation, and by the initial value of attributes. In GCM [66], a component definition may extend another component definition. It is possible to obtain different implementations of a component type by sub-typing its definition. A component can either be defined dynamically by means of the GCM API, or it can be defined statically thanks to the GCM ADL [66].

A component definition may also reference controllers, and it may define attributes that are dealt with by a specialized attribute-controller. The attribute-controller interface (see the example interface in Figure 4) must exhibit a
public interface anAttributeController extends AttributeController
{
    public String getValue();
    public void setValue(long value);
}

GCM ADL: <definition name="AComponent" >
<content class="AComponent"/>
<attributes signature="AComponentAttributeController">
    <attribute name="Value" value="10"/>
</attributes>
</definition>

Figure 4: Example of GCM attribute-controller definition and configuration.

public interface MeasurablePropertyController
{
    any getProperty(String property_name)
        throws IllegalPropertyException;
    List<String> getPropertyList();
}

Figure 5: Interface of a controller to retrieve any component property.

setter/getter behavior with respect to an attribute. Attributes can also be configured in the ADL as shown in Figure 4 for the Value attribute.

Dynamic component description. From our viewpoint, static information is not enough to accurately describe a component. Dynamic information about a component is also needed for component selection (either at deployment time or at connection time), as for example its execution time or the amount of memory used, and, conversely, this information may depend on the resource selected for deployment. Hence we need a mechanism to describe properties that need to be dynamically measured.

A component may export the values related to its dynamic behavior through attributes. This is a straightforward technique, but it suffers from two drawbacks. First, attributes are a general mechanism that targets component configuration: not all of them refer to a dynamic property of a component. Second, component implementations have to provide the implementation of the interface related to all attribute-controllers, while some measurable properties may only exist and be relevant while executing the component instance on a specific resource. It seems difficult to add an unplanned property to an existing component, thus attributes are an interesting but insufficient mechanism.

2.1.2 Measurable property definition

We define a measurable property as a value associated to a component instance that contributes to characterize it. The values of a measurable property obtained from two instances of the same component may be different as the value is measured on a component instance. Measurable properties are usually expected to be dynamically retrieved, and to depend on the computational resource where the component instance is executed.

Measurable properties that are associated to a component can be provided either by the component itself or by a framework. In the former case, properties are said to be internal, while in the latter they are said to be external. External properties are values which are more easily or conveniently computed outside the component, such as the memory or the bandwidth consumption.

Therefore, a mechanism is needed to associate a measurable property with its producer. We envision two different kinds of property producer. The simplest case is when an attribute provides the property. The second case occurs when the property is retrieved through an interface. These two situations are detailed below.

Internal properties are directly provided by the component itself. Hence, it appears straightforward to re-use the attribute mechanism to get them. As such values are read-only, only a getter method is required in the attribute-controller.

External properties are provided by the framework. Thus, the framework has to provide the values for several components. As the framework is not a component, we may only expect that it will provide an interface. With respect
to the GCM design philosophy, such an interface shall be exported through a component controller. Figure 5 gives an example of the API of such a controller. It is important to note that the implementation of such a controller must be the responsibility of the framework, not of the component.

When taking into account both internal and external properties, it is obvious that they can be combined into a coherent interface. With respect to the outside of the component, a unique mechanism is needed. Hence, a compliant component implementation shall provide a controller whose interface is of type `MeasurablePropertyController`. Its implementation shall be able to directly return the values of internal properties and it shall invoke the framework provided controller for external properties.

As shown in Figure 5, the `MeasurablePropertyController` interface also has an operation that returns the list of all properties available so as to cope with the introspection property of GCM. In general, it is not possible to statically know the list of properties that will be available once a component is going to be deployed. We decide to support the more general case. If needed, component definition can be extended to enforce the list of supported properties, but internal and external properties have to be differentiated so as to check whether the framework a component is going to be deployed to supports the requested external properties.

2.1.3 Time Features of Application Programs.

So far industrial Grid systems have not yet solved two important problems: devising a general and efficient methodology for the distribution of tasks for each concrete application, and efficiently planning the execution of complex Grid applications.

We want to address a major issue with respect to these problems, the general difficulty to know in advance the execution time for each application task. We will call this the problem of "inexact time". We do not address, however, the research on parallel performance models and techniques to derive execution parameters from bottom-up synthesis of analytic models.

We will instead exploit time-related measurable properties such as the known times required for the execution of programs on a certain set of machines, and eventually use them to approximate top-down the time depending on other parameters, like the number of available processors, the volume and quality (e.g. accuracy) of the data to be processed. The introduction of these resource-dependent and application-specific parameters into the set of properties characterizing application components, makes it possible to improve the efficiency of Grid applications.

In a methodology addressing application behavior estimation it is necessary to determine the time attributes during the development of application programs for the Grid. A prerequisite is that all application programs are thoroughly verified and tested in order to guarantee their quality, and avoid e.g. resource configuration related bugs. The lack of such a quality control procedure is one reason slowing down Grid adoption in business and industry.

To allow enhancing the planning of application executions, besides introducing dynamic measurable component properties, a modernization of Grid infrastructure is needed, to incorporate the related management services. As soon as the management centers have received the complete information about the resources required for one complex application at each stage of computation (including economic parameters), the execution can be planned with maximum efficiency. At the same time it will be possible to make corrections in the plan during the execution of an application depending on the current state of each Grid resource.

2.1.4 Interaction between Applications and the Grid

We address the efficient execution of complex Grid applications not only by solving the issue of elementary Grid jobs, but also with an appropriate Grid organization and with tools for the automated planning of the application execution exploiting a universal economical scheme of the distribution of resources. We will introduce a method for the organization of the management of complex applications which is based on the problem solving environment SEGL [48].

The planning process of a simulation experiment in a Grid environment covers all phases of the experiment from design and deployment through execution to the completion phase. At the beginning of the design phase, there will be an evaluation of the minimum resources required for the execution of the application. This evaluation has to take into account the execution time as given by the user. In addition the total number of resources required at each time will be calculated, taking into account the level of parallelization the application permits.

During the design phase the possible cost limits for the resources required for each block (as used in SEGL [49]) will
also be calculated. This includes that it will iterate budgets for each block depending on the total budget for the execution.

During the deployment stage planning will be taken over by the Management Center. In this phase the Deployment Module of the Application Manager makes use of various types of information services to get information like availability of resources, probability of failure, cost, and others. From this it will create a pool of domains and of individual candidate machines which are best suited for the execution of the given application.

Next, the management center acting on behalf of the users concludes preliminary contracts with the owners of the resources for the possible usage of Grid resources. After this, sub-server programs as already provided in SEGL representing the interests of the Management Center of the application are sent to the selected machines and domains. These sub-server programs will be responsible for the local monitoring of the current state of resources. They also act as agents during the conclusion of current and future transactions between the application and the owners of the resources. They also manage the execution of jobs.

As in the course of the execution the working condition of resources required is constantly monitored, the decision concerning their reorganization in the event of a failure can be taken on the spot. In such a case the old contract is canceled, a new set of resources is generated, and a new job is started.

Payment of resource is made according to the realization of jobs. The conclusion of contracts and the execution of payments is controlled by the bank of the resource owners and by the banks of the clients. The organization of the interaction between an application and the Grid in the deployment and execution phases is shown in Figure 3.

2.2 Positioning in the Institute activities

The contribution discussed here clearly fits in the activities of the Programming Model Institute as well as Institute for Grid Systems, Tools and environments.

We have proposed an extension of the GCM with measurable properties, to be managed by a combination of framework-provided interfaces and distributed retrieval services. The information provided about properties of component instances over different resources is leveraged in a testing and execution environment aiming at solving the “inexact time” problem. The approach allows the introduction into the Grid of traditional and transparent economic mechanisms for the resource allocation, both for the clients and for the resource owners.

The introduction into the Grid of an apparatus of management centers enables the realization at a high organizational level of an optimal planning and efficient control for complex Grid applications generated in different areas of science, industry and business.
3 Towards a spatio-temporal component model

by C. Perez, H. Bouziane (INRIA)

Grid infrastructures are undoubtedly the most complex computing infrastructures ever built incorporating both parallel and distributed aspects in their implementations. Although they can provide an unprecedented level of performance, designing and implementing scientific applications for Grids represent a challenging task for the programmers. One reason is that numerical simulation applications are becoming more complex involving the coupling of several numerical simulation codes to better simulate physical systems that require a multi-disciplinary approach. Component-based programming and service-oriented programming are two very popular candidates to design these applications using a modular approach. However, these two models promote two different composition logics: component programming appears as a spatial composition describing the connexion between components while service programming promotes a temporal composition expressing the scheduling and the flow of control between services. Let analyze these two orthogonal relationships.

A spatial composition is a relationship between entities such as components that are concurrently active during the time this relationship is valid. In general, components interact through ports, according for instance to the provides-uses paradigm. Hence, components must have adequate and compatible ports to be composed. For example, for a provides-uses composition, a component acts as a provider (resp. a user) if and only if it exhibits a provides (resp. uses) port. Then, the composition determines the direction of allowed interaction. In most spatial composition models, the direction is oriented: it is the user that invokes an operation on a provider. However, they do not inform on interaction frequency: it is not known whether the user will actually invoke an operation neither the number of invocations. It is only known that the components concurrently exist during the time the relation is valid, i.e. the components are connected. Consequently, a spatial composition enables to express the architecture of an application, typically captured by UML component diagrams[68]. Spatial composition principle is followed by most component models like GCM [66], FRACTAL [75], CCA [69], CCM [74], SCA [1], etc.

A temporal composition is a relationship between entities (tasks) that expresses an execution order of the entities. There are two classical formalisms for describing such kind of relationship: data flows and control flows. A data flow focuses on the dependencies coming from data availability: the outputs of some tasks $t_i$ are inputs of a task $T$. The execution of $T$ depends on the one of all $t_i$. In control flow models, the execution order of tasks is also given by some instructions. Most existing models provide instructions such as sequences, branches, loops, etc. Temporal compositions enable expressing the sequence of actions within an application which typically may be captured by UML activity diagrams[68]. There exist many environments that deal with temporal compositions such as workflow systems like ASKALON [67], TRIANA [71], KEPLER [72], BPEL4WS [73] or other cited in a taxonomy of Grid workflow systems provided in [70].

In some cases, spatial composition is very useful to describe the fact that some components must co-exist simultaneously and that they must exchange functionalities. It is the case for strong code coupling simulations like for example meteorological simulations. The main limitation of spatial compositions is that they do not explicitly capture the temporal dimension. As far as we know, ADL based component models are not able, through an assembly description, to express the fact that two components $A$ and $B$ do not need to be instantiated simultaneously because for example $A$ is a pre-processing with respect to $B$. From the ADL point of view, all components need to be instantiated during the application lifetime. It may lead to an underutilization of resources because of an overestimation of needed resources. Using an API to dynamically create/destroy components as can be done for example in CCA or GCM, partially solves the problem. A driver component can orchestrate components creation/connection/destruction. However, the drawback is that the composition is hidden in the code. Hence, any modification on the application structure requires to modify the code.

As for spatial composition, temporal composition improves code reuse by assembling black boxes. However, in contrast to spatial composition models, its main advantage is the enabling of efficient resources management thanks to the expressiveness of temporal dependencies. Nevertheless, the main limitation of temporal composition is the lack of support to express that two running tasks must communicate, as for example strong code coupling simulations. The solution of externalizing the loop of a code limits the coupling to coarse grained codes with respect to the overhead of launching a task.
### 3.1 Contribution: A spatio-temporal model based on GCM and AGWL

Our proposal is to combine spatial and temporal compositions into a coherent component model derived from GCM as well as from AGWL [67]. The requirements to support the temporal dimension within GCM are the introduction of the concepts of tasks, temporal ports, their management, and the expression of their composition. The proposal is based on choosing, reusing and potentially merging or extending the specification of components, ports, tasks and the composition model offered by GCM and/or AGWL as illustrated in Figure 7. Let briefly discuss some choice before presenting them in more detail. First, we name task-component a component that also supports the concept of tasks. Being a component, its definition is derived from the GCM component definition. Second, as task-components are based on GCM, the port model is based on the GCM one. Hence, it should be extended to support temporal ports. With respect to AGWL, data input and output ports seems to be a satisfactory solution to represent temporal ports. Third, to describe an application, our proposal aims to meet the level of expressiveness offered by a workflow language, in our case the level of AGWL. Hence, our approach is to start from this language and to extend it with missing concepts. It mainly consists in replacing activities with components and in introducing spatial ports and their connections. Fourth, such an extended AGWL will drive the life-cycle of task-components: the control flow will mainly determine it but with additional constraints with respect to spatial compositions. Let discuss these points.

#### 3.1.1 Extending GCM components with tasks and temporal ports

A component being defined by its ports, a new family of ports is needed to define a task-component. Let call them input and output ports. In contrast to classical client/server ports, that provide a method call semantic, input/output ports are attached to a data type. Existing workflow languages support many data types such as primitive types (int, string, etc), files, packages, etc. As GCM defines typed interfaces, our model follows the same logic, but on data. Hence, the GCM `TypeFactory` interface needs to be extended with an operation such as

\[
\text{TemporalPortType createFcTmpType(name, dataType, ...)}
\]

To be able to create an interface that manages data types and not interfaces.

The next step is to support a task within a task-component. A task can be viewed as a particular operation to be implemented by a user. How this operation is defined depends on several assumptions. For example, multi-task components required to define a triplet (task, inputs, outputs) for each task, while it may be implicit for single task-component. Because of lack of space and with no loss of generality, the support of only one task per component is presented hereafter. Once the inputs of a task are received, the task can be launched and once it finishes, output data should be sent to connected input ports. However, that should be the role of a framework. Hence, a task can be perceived from two points of view: a developer and a framework point of view. In the case of GCM, the framework role can be assigned to a dedicated controller (the stubs of the component). Hence, we extended the GCM specification with an interface `TaskController` that contains only one operation `void task()` that represents the interface to be implemented by a developer. Such `task` operation is called when all the data have been received on the input ports.

#### 3.1.2 Life cycle management of task-components

A component can be started, i.e. it can provide services and/or use external services through its server and client ports. Moreover a component can be executing a task. To be able to distinguish between these two states, we propose to re-use the standard `started` state for the former case and to introduce the `running` state for the latter.

---

**Figure 7:** GCM and AGWL concepts used for defining a spatio-temporal model.
3.1.3 A composition language based on a modified AGWL

The application composition model proposed is inspired from the AGWL language. The objective is to preserve its algorithmic composition logic but based on a task-component assembly view. Hence, the approach is essentially based on the replacement of the activity concept by a task-component one. In other words, we need to make AGWL GCM components aware. This section gives an overview of the impact of such a replacement according to the definition and composition of components. The term component refers to the GCM based task-component specified in the previous section.

<!-- AGWL activity -->
<activity name="name" type="type">
<dataIn name="name"/>*
<dataOut name="name"/>*
</activity>

<!-- AGWL sub-workflow -->
<subworkflow name="name">
<dataIn name="name"/>*
<body> <activity>+ </body>
<dataOut name="name"/>*
</subworkflow>

1 <!-- modified AGWL : component type part -->
2 <component name="name" (extends="parentType")?>
3 <dataIn name="name" type="dataType"/>*
4 <dataOut name="name" type="dataType"/>*
5 <clientPort name="name" type="interfaceName"/>*
6 <serverPort name="name" type="interfaceName"/>*
7 <attribute name="name" type="attributeType"/>*
8 <!-- other spatial port types -->
9 ( <impl type="exe|dll|.." signature="sign" /> |
10 | <body> <component>+ </body> )
11 <controllerDesc desc="desc"/>?
12 </component>

Figure 8: From an AGWL activity and sub-workflow to a GCM task-component.

Component definition The type of a component is essentially defined by its ports. The internal structure of a component (binary code for a primitive component and composition description for a composite) represents configurable parameters to be applied when a component is instantiated. A same component type can be configured to be primitive or composite and with different implementations. The two kinds of components are then viewed as a sole concept when they are defined. In contrast, an atomic activity type in AGWL is distinguished from a sub-workflow, as described on the left of Figure 8. This distinction does not seem to be a necessity, as an internal composition description can be viewed as a particular implementation. Therefore, our proposal keeps the GCM logic and replace both activity and sub-workflow constructs by a sole component concept. The result is shown in the right of Figure 8. A component definition may inherit from existing ones, and may specify temporal, spatial or attribute ports (from line 3 to line 8).

It can also configure its internal structure thanks to the impl (line 9) or body (line 10) elements for respectively a primitive and a composite components, as well as its membrane description (controllerDesc, line 11). The content of a body element is explained hereinafter.

Composition The next step is to determine the impact of a spatio-temporal composition on the composition principles of the AGWL language. In particular, we focus on port connection/configuration, data flow composition, control flow composition and spatio-temporal composition.

Connecting and configuring ports: A connection consists in configuring the value of a client or an input data port. The value of the configured port is respectively a reference to a server port or to an output data port. Figure 9 shows the enriched elements of a component definition for connection concerns. Two possibilities are offered to the user to configure a port. A port can be configured when it is defined, thanks to the set attributes (lines 3 and 5). Ports can also be configured latter. That is done thanks to the setPort instruction (lines 11 and 12) which is used to connect spatial and temporal ports. An attribute port (line 7) is a particular client port which does not require a connection. From the point of view of AGWL, the connection logic is almost the same. The sole difference is the addition of spatial ports connections.

Data flow composition: As connecting an input data port to an output one is a direct mapping of a data dependency specification between tasks, data-flow composition remains the same as in AGWL.

Control flow composition: In order to compose an application according to a control flow composition, our approach is to respect the composition principle offered by AGWL. That means, a control construct (sequence, if, while, etc.) is considered as a special kind of white-box components. The particularity is that the internal structure of these components is pre-defined as well as an associated data-flow model for validity checking. A control
structure can nevertheless define classical ports and can be reused as a component. Figure 10 presents two AGWL control structures adapted to our proposal. Compared to the component definition shown in Figure 9, the body of the sequence component is implicit. It is determined by all its internal components: the sequence is implied by the order of declaration. The two bodies of the if construct are however explicitly delimited by the then and else elements. Line 11 allows pre-declaring components for which the objective is explained below. As control structures appear as components, they can be reused everywhere within a component body.

Spatio-temporal composition: With respect to the component specification shown in Figure 9, spatial, temporal and spatio-temporal compositions can be described. In addition, it is possible to connect ports, in particular spatial ports, at different levels of a composition. However, when components are involved in a spatio-temporal composition, it may be required to determine the expected behavior. To overcome confusion and to enable a composition that reflects as much as possible the suited behavior, we propose to define a simple priority system: if a spatial connection is specified within a control structure body then, the temporal dimension is prevailing, otherwise the spatial dimension is to be considered first. Therefore, if two spatially connected components are involved in the temporal composition of an application and if only one task belonging to one of two components is reachable by the control flow then, in the first case, the composition is considered illegal.

3.2 Positioning in the Institute activities

The institute of the Programming Model Institute is working on defining GCM, a grid-enabled component model. Moreover, an horizontal activity related to Services has been started. As both models have benefits and drawbacks with respect to some algorithmic patterns, this paper explores the possibility of designing a model that support both composition models. As a result, it describes a model based on two existing models – GCM as a component model and ASKALON as a workflow model. We made the decision to extend GCM with temporal ports and task concepts and to adapt AGWL to offer a spatio-temporal composition language.
4 A component platform for experimenting with autonomic composition  

by L. Henrio, D. Caramel, F. Baude (INRIA)

The autonomic computing paradigm [36] was inspired by the (complex) human nervous system. Generally speaking, autonomous applications implement complex management strategies through a decentralized independent decision process. Their goal is to ensure the self-* properties [36], and more generally all the self-management features. Those management strategies themselves can be considered as non-functional aspects. For this reason, many researchers claim that it would be very useful for designers and developers of complex autonomic strategies to have a clear separation between the functional and non-functional aspects of the application. To this aim, it is also much easier to work with a clear representation of the functional and non-functional architecture of the application. We propose a programming model and a framework which brings solutions to potentially further ease the development of complex autonomic strategies. Our solution is grounded in a component-oriented approach to develop autonomic applications. Besides, our intention is also to enable others to use this platform as a mean to easily experiment new autonomic behaviours: this requires to be able to quickly design and program new behaviours, so the capability to reuse already developed features is a strong requirement. Also, as some autonomic behaviours one may want to experiment with may be inspired by nature (as conducted for instance within the EU funded BIONETS\(^1\) research project we are involved in), we need to provide a framework through which similar but numerous participants must be modelled and then emulated. As sometimes in nature, self-* properties may result from evolution, so a second requirement is that the autonomic behaviours are designed and programmed in a way that permits them to modify themselves even at runtime. Finally, we also foresee that emulations be computation intensive, so we require to be able to deploy and run the platform on a sufficiently big aggregation of computation resources like computing grids are, without additional burden for the experimenter. This is why the component-based framework we propose for experimenting with autonomic behaviours is grounded upon a software component model initially dedicated to the programming of Grid applications: the Grid Component Model (GCM) [30]. Besides, our effort contributes to easing the programming of autonomic Grid applications (which is an active current research track by itself, see e.g. [43, 21]).

Our purpose in this work is not to provide algorithms for autonomic strategies but to provide support for such algorithms, meaning control on the non-functional aspects of a component system, possibility to plug dynamically different management strategies, and runtime support for autonomic systems . . .

A component is a software module, with a standardized description of what it needs and provides (called server (provided) and client (required) interfaces), that can be manipulated by tools for composition and deployment. Interfaces can be connected (bound) together to allow components to interact, and constitute a component assembly.

From a practical point of view, the component model we rely on is GCM. This model has usual advantages of component models (structure, hierarchy and encapsulation) and offers some reconfiguration primitives (bind, unbind, add and remove components). One of the strong advantages of this model is to represent both the functional and the non-functional parts of the application as a component system. This allows to easily design complex autonomic strategies. Autonomic strategies can be designed as a component system belonging to the non-functional part of the application. By using reconfiguration possibilities of such a system, these strategies can be dynamically updated. Moreover, the model has been extended to allow consistent (well-defined) interactions between non-functional and functional parts.

4.1 Contribution

4.1.1 Programming Autonomic Applications

Autonomic computing is a paradigm that proposes to add to software entities some autonomous capabilities: the entity is capable to self-adapt in reaction to context or environmental changes. Adaptation is generally designed and described in an ad hoc way, which involves trying to predict future execution conditions at development time and embedding the adaptation decisions in the application code itself. This approach has several drawbacks: increased complexity (business logic polluted with non-functional concerns) and poor reuse of software caused by a strong coupling with a specific environment. As previously noticed in the literature (e.g. [32]) adaptations (most notably those related to resource usage) can be decoupled from pure functional concerns. This approach does not have the same drawbacks as the ad hoc way. We believe that application developers should be able to concentrate on pure business

\(^1\)Bionets EU project (IST-FET-FP6-027748) at www.bionets.eu
logic, and write their code without worrying about the characteristics and resource limitations of the platform(s) it will be deployed and run on. Then, the adaptation logic, which deals specifically with the adaptation concern, is added to this non-adaptive code, resulting in a self-adaptive application able to reconfigure its architecture and parameters to always fit its evolving environment.

**Conditional expression / Rule based** As mentioned in [42], the spectrum to express self-adaptability is broad. At one extreme (bottom) lie conditional expressions in the form of If condition/then action rules. In its simplest form, this methodology is not very flexible (selects among predetermined alternatives), only supports localized changes because it would be difficult to unwind the control loop in a synchronized manner on all – possibly remote – software entities that are concerned at the same time, and lacks separation of concerns.

**Aspect Oriented Programming** Aspect oriented programming [37] has been designed to allow separation of concerns in the design of an application. Embedding “autonomic rules” as aspects allow a better re-usability of the programs: different aspects can be designed corresponding to different deployment environments and can be freely composed with different business applications. Concerning dynamic evolution however, pure aspect-oriented approach is still quite limited. Indeed, aspects are usually weaved within the functional code at compilation or instantiation time, which prevents this non-functional code from being modified at runtime. One has to weave again the adaptation code when deploying on a new infrastructure. Such restriction w.r.t. runtime weaving motivated the introduction of Dynamic Aspect-Oriented languages, and in particular their application to autonomous systems [34].

**Autonomic Distributed Components** Component-based development has emerged as an effective approach to building complex software systems; its benefits include reduced development costs through reusing off-the-self components and increased adaptability through adding, removing, or replacing components. This is why component programming frameworks are becoming attractive in networking [24], in large-scale distributed computing (a.k.a Grid computing) be it dedicated to scientific computing [26] or enterprise computing [27], in mobile and situated autonomic communications [39], and more generally in any running context constituted of fixed or intermittently connected devices. In general, the followed approach for autonomic components consists in wrapping around each component an autonomic manager [43, 22] that is guided by the interpretation of some rules or contracts [38, 23, 41] dependent of the requested self-properties, and possibly injected at runtime. The actions that those rules trigger may encompass reconfiguring the component-based application. All these actions pertain to monitoring the base application and at the same time, take into account non-functional concerns due in particular to the running context. Besides, it appears that component and aspect-oriented approaches can complement each other very well [44], because at some point, non-functional concerns may be dependent or have impact to functional ones. So techniques from AOP can be relevant to be used. AOKell implements a non-distributed version of the Fractal [28] component model.

### 4.1.2 Structuring Non-functional Concerns with Components

An application can be split between some functional code implementing the business features, and some non-functional code for managing the application, and supporting its execution. In Fractal and GCM component models, the non-functional (NF) part of the components is called the membrane. It is composed of controllers that implement non-functional concerns. During their execution, components running in dynamically changing execution environments often have to adapt to these environments. The membrane of Fractal/GCM components is the adequate location to host adaptation strategies, which in theory can be as complex as needed, i.e. completely autonomic. When the behavior of a strategy is not optimal, it has to be updated or changed dynamically. To do this, the controllers architecture has to support recomconfigurations at runtime, which is the case of a component system. Examples of use-cases include changing communication protocols, updating security policies or taking into account new runtime environments in case of mobile components. Adaptability and autonomicity imply that evolutions of the execution environments have to be detected and acted upon. They may of course imply interactions with the environment but also with other components for achieving management strategies.

We provide tools to plug and dynamically reconfigure autonomic strategies inside the membrane. For this, we provide a model and an implementation, using a standard component-oriented approach for both the application (functional) level and the control (NF) level. Having a component-oriented approach for the non-functional aspects also allows them to benefit from the structure, hierarchy and encapsulation provided by a component-oriented approach. This has already been adopted or advocated in [40, 35]. The solution that is suggested by the GCM is to allow, like in
Figure 11: Structure for the membrane of Fractal/GCM components

[44, 40], to design the membrane as a set of components that can be reconfigured. The GCM description[30] suggests the possibility to implement the membrane as a set of components. [25] goes more into details and suggests a structure for the composition of the membrane and an API for manipulating it. In GCM, non-functional components that are included inside the membrane can be distributed, just like the functional ones.

In order to be able to compose non-functional aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality, and contingency. For example, comparatively to Fractal, the GCM adds client non-functional interfaces to allow for the composition of non-functional aspects and reconfigurations at the non-functional level.

4.1.3 Plugging Autonomic Behaviours Inside Components

The GCM can encompass two kinds of autonomic behaviours: the one consisting in (autonomously) adapting a component to its changing environment; and the one consisting in (autonomously) adapting the components to evolving user-requirements for the applications (generally concerning the quality of services).

Autonomic Components = GCM Components + Autonomic Managers Inside Membranes  Our proposal for autonomic control of components is to encapsulate a set of manager components inside the membrane of each component. These components can be easily changed dynamically depending on the environment or the evolution of the component system. Interface and typing of components allow those autonomic managers to be bound together in a very structured way, which enables the distribution of the decision process: each autonomic manager can be responsible of the management of its component, and still communicate with other managers or with the external world if necessary. Moreover, hierarchical structure of components help managing autonomously a set of components and scales better: a decision process can be taken by delegating sub-decisions to sub-components of a composite one.

4.1.4 Autonomic Distributed Services

The next step is to consider a network with low connectivity where entities involved in the computation can be disconnected at any time. This demanding context underlines the limitations of the classical component design of applications: components composed together are assumed to correspond to runtime entities that are to be involved in a computation, and thus available.

Low Connectivity  When one switches to distributed computing environments featuring low connectivity as those relying on a mobile ad-hoc network, this is not sufficient. Indeed, first disappearing nodes (and consequently, disap-
pearing services that were running on those nodes) can appear afterwards. Moreover, the network is not structured enough to allow the creation of a unique new entity replacing the missing one easily. Consequently:

- Any binding can disappear at any time, sometimes reappearing later on;
- When a service is needed, it might have to be discovered at runtime according to a description of the desired service;
- Some services may appear, providing new functionalities.

**Better Decoupling Communications** A first aspect to consider in applications resulting from the composition of distributed services or components is the way communications are performed. As highlighted by [31, 33], only few efforts have been devoted to programming models for applications that must take for granted that they rely on such a low-coupled and transient communication mode. Their communication model is quite similar to the ASP calculus presented in [29]. However, AmbientTalk programming model is slightly different because there is no blocking synchronization, whereas ASP relies on a more coupled programming principle allowing to write programs more deadlocks prone but in a more intuitive manner. Both of those programming models allow a loose coupling of components, which is necessary for autonomic distributed services.

**Composition Plan:** A composition paradigm releasing the strong connectivity constraint of components No matter which programming model is chosen, we argue that, at the composition level some features have to be provided to better compose the basic blocks, resulting in a low-coupled application. Indeed, programming low-coupled applications easily results in a lot of code for dealing with the coupling process itself because this process is dynamic due to the possible volatility of those basic blocks, making difficult and error-prone the basic functional programming. Our objective is to be able to propose a composition method for such applications following a component-like approach: taking advantage of its qualities concerning structure of the program and high re-usability of the components; but authorizing low-coupling in the sense that some of the components may disappear during the process, and (if it is compatible with the purpose of the application) some other components fulfilling the same service may be found to replace them dynamically. Thus, we choose to rely on components which are composed in an abstract way and so provide a structure for the application that we can consider as an execution plan (a specification), which is structured because it is hierarchical; but we require a strong separation between application specification and implementation, allowing at runtime entities fulfilling the component description to be discovered, and to disappear. As services appear and disappear, a definition of a component involved in a composition may be too precise, i.e. too constraining. Consequently, the framework must be able to discover services based on a partial description, or even replace a service whose description does not correspond to any available service by a different service resulting from a dynamic (possibly hierarchical) composition of some more elementary but currently available services.

The stability of composition is completely lost in such a world, but this is partially compensated by the stability of the execution plan: the execution plan never disappears. The composition plan is a stable plan, and can be considered as the specification of a service.

To conclude, this new notion of component composition is particularly adapted to loosely coupled systems, because it does not rely on stable bindings between composed entities. For example it would be particularly well fitted to the design of a component model for Ambient-oriented programming.

### 4.2 Positioning in the Institute activities

This work is related to the institute activities by both its extension, implementation, and refinement of the GCM programming model, but also by its illustration of possible impacts of the GCM: how the GCM can be used to develop generic frameworks allowing autonomic scenarios to be implemented. Moreover, we have shown a possible extension of the GCM for providing loosely coupled components.

#### 4.2.1 Extensions, Implementation, and Refinement of the GCM

As illustrated in Fig. 11, we refined the component model defined in the GCM specification concerning the “dynamic controllers” we refined and implemented component controllers and the associated interfaces. This implementation comes with a precise description of the possible non-functional interfaces a component can have, and their possible
bindings. This refined model will ease the future implementations of the GCM providing component controllers, which as shown here are mandatory in order to provide autonomic applications.

4.2.2 A General Framework for Autonomic applications, based on the GCM

Our contribution also illustrates application domains for the GCM. GCM has always been designed taking into account autonomicity, but this work provides a new point of view on this aspect. Any part of the component assembly can entirely evolve in an unplanned way, including the component controls, and even the autonomic strategy. Finally this framework seems to be the perfect context for experimenting with autonomic strategies.

We hope this aspect will provide the institute with new applications for our programming models.

4.2.3 A Loosely Coupled GCM Model

We also investigated here how the GCM model could be adapted in the context of loosely coupled entities and unreliable connection between components, thus extending the impact of the GCM and, we hope, the impact of the programming model institute on distributed programming, going further than the domain of Grid computing.
5 Deductive verification of GCM: Deontic Temporal Resolution
by A. Bolotov (UOW)

5.1 Contribution

5.2 Positioning in the Institute activities
6 Experiments with GCM and SCA: preliminary experiences and results
by M. Danelutto, G. Zoppi (UNIPI)

Recently algorithmic skeleton concepts introduced by Murray Cole in late ‘80 [9] moved to the grid scenario and
where used to program several different kind of super/meta/skeleton components modelling common grid program-
ming paradigms [59, 7, 4]. Components embedding of skeletons somehow filled the gap among skeletons concepts
and software engineering concepts. As a matter of fact, component technology allowed to embed skeletons in well
know programming paradigms (the components) and to hide non relevant implementation parameters to final skeleton
users (by exploiting hierarchical component composition facilities). Eventually this allowed more and more high level
programming abstraction to be provided to application programmers. In component frameworks, skeletons are usually
implemented through composite components and basically provide the very same programming pattern that classical,
non component skeleton programming environments usually provide to the application programmers. Skeleton code
parameters were provided as components, allowing hierarchical and incremental program development. Component
technology simplified somehow code deployment on remote processing elements leveraging onto component frame-
work facilities rather then requiring explicit and consistent programming efforts in the design and implementation of
the skeleton compiler and run time tools.

After successfully migrating to the grid scenario via component technology a further step was made: skeletons
where used to combine powerful parallel application pattern abstraction with typical grid related autonomic manage-
ment features. Behavioural skeletons were thus introduced in 2007 [51, 6] in the framework of the CoreGRID and
GridCOMP projects. A behavioural skeleton is basically a skeleton integrated with an “autonomic manager” item
taking care of all the non-functional aspects related to skeleton implementation, such as performance optimization,
fault tolerance and security. The autonomic managers, in this case, can be understood as the locus where well known
self-optimization and self-healing autonomic [61] features are implemented.

In the meanwhile, grid programming environments evolved more and more through (Web) Services and web
services [20] become a de facto standard in several related scenarios: grid programming, enterprise applications, sw
interoperability, incremental application development, etc.

In this contribution we basically discuss the results achieved by implementing a specific GCM behavioural skeleton
(the functional replication/task farm skeleton as described in [6]) on top of SCA (the Service Component Architec-
ture introduced by IBM [1]). By this experiment we aimed to follow Cole’s “manifesto” suggestion to propagate the
concept with minimal disruption [10]. The task farm skeleton discussed in this works provides service application
programmers with a very high level programming paradigm that can be easily used to program most of the typical em-
barrassingly parallel grid/distributed applications actually leaving to the component implementing the programming
paradigm (to its autonomic manager, actually) the hard task to implement self optimization and self healing features.
We also followed Cole’s recommendation to accommodate diversity by providing user friendly ways of extending and
modifying autonomic management features and policies. Last but not least, implementation of behavioural skeletons
on top of SCA allowed a comparison to be performed with the already existing similar implementation of behavioural
skeletons on top of ProActive/Fractal [17] developed within GridCOMP [64].

6.1 Contribution

Service Component Architecture [1] provides the user with a programming framework supporting application de-
velopment based on Service Oriented Architecture. Applications programmers may build their application re-using
existing services embedded in service components and specifying composite components through proper XML files.
Eventually, SCA applications can be run on several distinct kind of distributed platforms exploiting existing technolo-
gies such as web services or RMI. SCA bindings are provided for different programming languages in such a way
programmers can use (among the others) Java and C++ code to program the primitive SCA components.

Typical SCA component assembly is the one of Figure 12 (the Figure is taken from SCA documentation). In this
case, the composite itself is specified using an XML file detailing all the internal components and wires as well as all
the external use/provide interfaces. The composite can be run on the SCA runtime as well as on plain web services
runtimes. Several alternative methods can be used to export and reference the component ports.

The big advantage of SCA framework relies in the fact it provides very handy ways to build applications from
existing services (which is the very same thing you can do using BPEL in other context) and allows the service
composition to be (re-) used within applications as a primitive component.
In order to write a Java SCA program, programmers must basically perform the following steps. First, interfaces declaring the component interface should be provided. In the interface services (methods) as well as properties (instance variables) provided by the component are declared. Then Java code/classes implementing these interfaces should be provided. Eventually, XML files describing composition of components have to be written, declaring the component name, what the component exports (provide ports), uses (use ports) and how the component are implemented (java classes used). Once these steps have been performed, the program may be launched by instantiating a SCADomain and passing it as a parameter the XML .composite file hosting the component composition specification. At this point the component can be accessed by clients querying the SCADomain a reference to the component (this is achieved using the name provided in the XML file) and accessing the component exported features. Again, the interested reader will find all the details relative to SCA on the web site hosting all the documentation [1]. We used here the Tuscany open source implementation of the SCA service component framework [3].

6.1.1 Implementation

Our implementation of the task farm behavioural skeleton was designed as shown in Figure 13. A WorkpoolService composite SCA component has been implemented and it is provided to the user that completely takes care of implementing a task farm, with respect to both functional and non functional behaviour. The WorkpoolService component exposes interfaces (provides methods) to submit jobs (and this is a functional concern) as well as to start autonomic management and to submit rules affecting autonomic management behaviour (this is instead a non functional concern). Autonomic management is implemented exploiting JBoss rules [2]. Each Jboss rule includes a precondition as well as the actions to be executed in case the precondition is satisfied. Both preconditions and actions use proper Java beans associated to the entities managed within the WorkpoolService. Methods of these beans may be invoked within a JBoss rule in order to achieve some (part of an) autonomic self.* behaviour. The JBoss rules associate to a SCA task farm behavioural skeleton component are activated only in case the user explicitly asks to start autonomic control management. Some predefined rules, such as increase the parallelism degree in case the task pool service time happens to be larger than user defined service time contract, are predefined in the task farm component. Other rules can be defined on-the-fly by the task farm component user and inserted via its non functional interfaces.

The TaskManager component inside the WorkpoolService actually takes care of tasks submit requests and uses one of the WorkerManager components to execute the submitted task, in such a way parallel execution of the submitted task stream is achieved. Each one of the WorkerManagers take care of the Worker components allocated on the same resource (processing element) used to run the WorkerManager. In turn, Worker components in the resource are allocated instantiating copies of the Worker components provided by the user through the WorkpoolService func-
tional interface. Each Worker is able to compute a single submitted task at a time. The WorkerManager component provides a functional submitTask interface as well as non functional addWorker and deleteWorker interfaces that can be used, upon WorkpoolService requests, to allocate and deallocante computing components on the resource. The WorkpoolService basically includes the autonomic management of the task farm behavioural skeleton as described above.

Overall the WorkpoolService implements a component that can be used to compute stream of tasks in parallel. The amount of resources allocated in the component (i.e. the amount of Worker components used) is dynamically decided according to the JBoss rules programmed in the component. By default, a number of Worker components will be eventually allocated that allows to achieve a service time (time incurring among the delivery of two results relative to the computation of two different submitted tasks) which is less or equal to the interarrival time of the submit requests.

6.1.2 Experimental results

We run some experiments on a cluster of Linux workstations interconnected through a Fast Ethernet network to verify the functional features of our SCA task farm behavioural skeleton as well as the non functional features of its autonomic management.

First of all, scalability has been measured. Fig. 14 left shows typical scalability curves we got when running coarse grain (i.e. tasks taking significantly longer (100 times) time to compute than to get the input data and to deliver output results) tasks thought the task farm SCA behavioural skeleton. It is not surprising that coarse grain tasks achieve almost perfect scalability as the tasks are independent (task farm implements an embarrassingly parallel computation pattern). In case the task computational grain is lower (O(10) instead of O(100)) the task farm behavioural skeleton stops scaling at about 8 workers.

We then provided JBoss rules to adapt the task farm behavioural skeleton to varying performance achieved in the execution of the user tasks and we explicitly activated autonomic management. Fig. 14 right shows what happened when rules were used stating the service time of the task farm should be kept smaller than a given value and additional load was put on the resources used to compute the tasks. The autonomic manager inside the WorkpoolService detected a decrease in the service time and started increasing the parallelism degree until the required service time was obtained again. In this case, additional load was deployed on the first four worker resources when about 500 tasks (out of 1K tasks) were computed. The task farm behavioural skeleton reacted autonomically and started new workers on four additional computing nodes, whose WorkerManager had no Worker allocated up to that moment. The whole thing happened without any explicit programmer intervention, but supplying the proper JBoss rule stating that in case the service time goes down under a given threshold, new workers should have been added, such as

```
rule "AdaptUsageFactor"
  when $workerBean: WorkpoolBean(serviceTime > 0.25)
  then $workerBean.addWorkerToNode("";
end
```

CoreGRID TR-XXX 23
Figure 14: Scalability (left) and effect of dynamic management (right) in task farm behavioural skeleton

After implementing the task farm behavioural skeleton on top of SCA Tuscany, we verified that all the necessary mechanisms are present in SCA. The only thing we had to emulate programmatically was dynamic composite component modification, by generating proper .composite files that are then used to instantiate the new (modified) composite components.

6.2 Positioning in the Institute activities

The contribution discussed here clearly fits in the activities of the Programming Model Institute related to Task 3.3 “Advanced Programming Models” as well as on the horizontal activities related to Services.

The “Advanced Programming Model” task investigates the possibility to implement programming environments on top of GCM that contribute to raise the level of abstraction presented to the grid programmers, and therefore to increase the efficiency of the grid application development process. The idea of providing the programmer with a component/service embedding and handling all the non-functional aspects related to the implementation of a task farm pattern goes in the direction of providing a more complete structured parallel programming environment providing components/services modelling all the useful, reusable and efficient parallelism exploitation patterns usually exploited within grid applications.

SCA exploitation, on the other hand, has a twofold effect: on the one hand, by migrating GCM concepts to the Service world, we contribute to disseminate the GCM methodology and philosophy in a context that has not yet been addressed by the Programming Model Institute activities. On the other hand, the availability of complete, customizable and autonomic task farm service contributes to show how services may be used to implement complex orchestration patterns, supporting grid application development.
7 Advances in GCM

by N. Tonellotto, P. Dazzi (ISTI/CNR),
M. Aldinucci, S. Campa, M. Danelutto (UNIPI)

7.1 Contribution

Typical grid architectures are subject to dynamic changes that impact their behaviour [60]. As a consequence, grid applications need to dynamically adapt to the features of the underlying architecture in order to be efficient and/or high performance [52]. In recent years, several research initiatives exploiting component technology [56] have investigated the area of component adaptation, i.e. the process of changing the component for use in different contexts. This process can be dynamically defined by allowing adaptations, in the form of code, scripts or rules, to be added, removed or modified at run-time. The idea of autonomic management of parallel/distributed/grid applications is present in several programming frameworks, although in different flavours. ASSIST [65, 52], AutoMate [63], SAFRAN [57], K-Components [58] and finally the forthcoming CoreGRID Component Model (GCM) [56] all include autonomic management features. GCM is a hierarchical component model explicitly designed to support component-based autonomic applications in highly dynamic and heterogeneous distributed platforms, such as grids. It is currently under development by the partners of the EU CoreGRID Network of Excellence. A companion EU STREP project, GridCOMP is currently developing an open source implementation of GCM and preliminary versions are already available for download as embedded modules in the ProActive middleware suite.

Autonomic management aims to attack the complexity which entangles the management of complex systems (as Grid applications are) by equipping their parts with self-management facilities [61]. GCM is therefore assumed to provide several levels of autonomic managers in components, that take care of the non-functional features of the component programs. Each GCM component contains an Autonomic Manager (AM), interacting with other managers in other components. The AM implements the autonomic cycle via a simple program based on reactive rules. In this, the AM leverages on component controllers for the event monitoring and the execution of reconfiguration actions. In GCM, the latter controller is called the Autonomic Behaviour Controller (ABC). This controller exposes server-only non-functional interfaces, which can be accessed either from the AM or an external component that logically surrogates the AM strategy.

Programmers may write their own AM and ABC implementation using the mechanisms provided by the GCM run time. This requires substantial knowledge on the part of programmers, relating to both autonomic control principles and to the component model itself. Without such detailed knowledge it is very difficult to develop efficient and effective autonomic controllers/managers. We recognise, however, that common patterns of autonomic management can be adopted in grid applications, and, to this end, we have introduced behavioural skeletons [51].

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

They represent a specialisation of the algorithmic skeleton concept for component management [55]. 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 skeletons’ inherent 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 [59]). 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 a) exposes a description of its functional behaviour, b) establishes a parametric orchestration schema of inner components, c) may carry constraints that inner components are required to comply with, and d) may encompass a number of pre-defined plans to cope with a given self-management goal.

Behavioural skeleton usage helps designers in two main ways. First, the application designer benefits from a library of skeletons, each of them carrying several pre-defined, efficient self-management strategies. Then, the component/application designer is provided with a framework that helps both the design of new skeletons and their imple-
In both cases two features of behavioural skeletons are exploited: on the one hand, the skeletons exhibit an explicit higher-order functional semantics that delimits the skeleton usage and definition domain. On the other hand the 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.

![Functional replication behavioural skeleton schema](image)

Figure 15: Functional replication behavioural skeleton schema

In [51] we introduced a simple set of behavioural skeletons, mainly modelling functional replication parallel patterns. We assumed our skeletons have two functional interfaces: a one-to-many stream server S, and a many-to-one client stream interface C. They accept requests on the server interface and then dispatch the (partial) requests to a number of instances of an inner component W which may propagate results outside the skeleton via C interface (see Figure 15). We assume that replicas of W can safely forget 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.

A notable instantiation of behavioural skeletons exhibiting functional replication is task farm. A task farm processes a stream of tasks \( \{x_0, \ldots, x_m\} \) producing a stream of results \( \{f(x_0), \ldots, f(x_m)\} \). The computation of \( f(x_i) \) is independent of the computation of \( f(x_j) \) for any \( i \neq j \) (the task farm parallel pattern is often referred to as the “embarrassingly parallel” pattern). The items of the input stream are available at different times, in general: item \( x_i \) is available \( t \geq 0 \) time units after item \( x_{i-1} \) was available. Also, in the general case, it is not required that the output stream keeps the same ordering as the input stream, i.e. item \( f(x_i) \) may be placed in the output stream in position \( j \neq i \).

In this case, in our farm behavioural skeleton, a stream of tasks is absorbed by a unicast S. Then each task is computed by one instance of W and the result is sent to C, which collects results according to a from-any policy. This skeleton can be equipped with a self-optimising policy as 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. Therefore, the AM just checks the average time tasks need to traverse the skeleton, and possibly reacts by creating/destroying instances of W, and wiring/unwiring them to/from the interfaces.

### 7.2 Implementation

In the case of the farm behavioural skeleton, the reconfiguration operations require the addition/removal of workers as well as the tuning of distribution/collection strategies used to distribute and collect tasks and results to and from the workers. The worker addition/removal operations can be used to change the parallelism degree of the component as well to remap workers on different processing elements and/or platforms. The distribution/collection tuning operations can be used to throttle and balance the resource usage of workers, such as CPU, memory and IO. The introspection operations involve querying component status with respect to one or more pre-defined QoS metrics. The component status is generally obtained as a harmonised measure involving component status and inner component status.

In the following we describe in some detail the implementation of a reconfiguration operation.

**add_worker**( k )  
**Semantics:** Add k workers to a skeleton of the functional replication family.

1. **Stop.** The ABC requires the Lifecycle Controller (LC) to stop all the components. To this end, the LC retrieves from the Content Controller (CC) the list of inner components \( W_1, \ldots, W_n \), and then issues a stop on them.

2. **Type Inspection.** All the \( W_1, \ldots, W_n \) have the same type. The ABC retrieves from the CC the list of inner components \( W_1, \ldots, W_n \), then retrieves TypeOf(\( W_1 \)).
3. **New.** One or more new inner components of type $\text{TypeOf}(W_1)$ are created.

4. **Bind.** The component server interface $S$ is wired to newly created $W_n, \ldots, W_{n+k}$ inner components via the Binding Controller (BC). $W_n, \ldots, W_{n+k}$, in turn, wire their client interfaces to the component collective client interface $C$. The process requires the inspection of the types of the interfaces of $W_1$ that is used again as a template for all $W_i$.

5. **Restart.** The ABC requires the LC to re-start all the components.

6. **Return.** Return a failure code if some of the previous operations failed (e.g. inner components do not implement stop/start operations); return success otherwise.

Experiments have been conducted on the current prototype of the GCM that is under development in the Grid-COMP STREP project [64]. The experiments mainly aim to assess the overhead due to management and reconfiguration. For the sake of reproducibility, the experiments have been run on a cluster instead of a more heterogeneous grid. The cluster includes 31 nodes (1 Intel P3@800MHz core per node) wired with a fast Ethernet. Workers are allocated in the cluster in a round robin fashion with up to 3 workers per node (for a total of 93 workers). Note however, the experimental code can run on any distributed platform supported by the ProActive middleware.

Figures 16, 17, and 18 respectively show the time spent on the farm behavioural skeleton for the stop, new and restart ABC services described above. This time is application overhead, since none of the workers can accept new tasks during the process. In the figures, a point $k$ in the X-axis describes the overhead due to stop/new/restart in the adaptation of the running program from a $k$ to $k + 1$ worker configuration. As highlighted by the curves in Fig. 16 and 18 the overhead of stop and restart is linear with respect to the number of workers involved in the operations.
This is mainly due to a linear time barrier within the Life cycle Controller (LC), which is an inherent part of the underlying ProActive middleware. Note that adaptation process does not strictly require such a barrier. Both stopping all the workers and linear time synchronisation are peculiarities of the current GCM implementation on top of the ProActive middleware, and not of the farm behavioural skeleton, which can be implemented avoiding both problems. Figure 17 shows the time spent for the new ABC operation (see Sec. 7.2). Again, in this case, the time is overhead. The experiment measures the creation of a single worker, and thus the times measured are almost independent of the number of workers pre-existing the new one. As highlighted by the Fig. 17 and 18 the overhead of the new and restart operations is much higher in the case where a fresh platform is involved (number of workers less than 32). The difference is mainly due to the additional time for Java remote class loading.

The results of the last experiment are presented in Fig. 19. It describes the behaviour of the application over quite a long run that includes several self-triggered reconfigurations. The application is provided with a QoS contract that enforces the production of a minimum of 1.5 results per second (tasks/s). During the run, an increasing number of platforms are externally overloaded with an artificial load (C++ compilation). The top half of the figure reports the measured average throughput of the filter stage, and the QoS contract. The bottom half of the figure reports the number of overloaded machines along the run, and the corresponding increase of workers of the filter stage. Initially the throughput of the filter stage is abundantly higher than requested ($\sim 3.5$ tasks/s); but it decreases when more machines are overloaded. As soon as the contract is violated, the AM reacts by adding more workers.

7.3 Positioning in the Institute activities

Autonomicity is one of the key points of the GCM component model. Behavioural skeletons provide the programmer with the ability to implement autonomic managers completely taking care of the parallelism exploitation details by simply instantiating existing skeletons and by providing suitable, functional parameters. Behavioural skeletons model fits perfectly into institute activities, in particular in the context of the task 3.3 (Advanced Programming Models).

With the present contribution, we discussed how behavioural skeletons can be implemented in the framework of the GCM component model. In particular, we analysed several issues related to the implementation of a functional replication behavioural skeleton. We presented experimental results that demonstrate both the typical overheads involved in autonomic management operations and also dynamic adaptation occurring during execution of a long-running application. Indeed, behavioural skeletons simplify the GRID components programmability and enhance code reuse with respect to the autonomic management issues. The model has empirically demostated to be able to manage the issues that mainly derive from the features that are peculiar to GRID, namely heterogeneity and dynamicity.
8 Secure component composition and definition with QoS

by J. Cohen, S. Mc Gough (IC)

This document covers material that was presented at the CoreGrid WP3 plenary meeting on Thursday 17th January in Paris. The presentation was intended as a summary of the activities and tasks which have been conducted at Imperial College London through CoreGrid-related projects in the period from 2000 until the present day. This has included projects such as the Imperial College E-Science Networked Infrastructure (ICENI) [91, 92, 93], the European Union funded GRIDCC project in which we worked on workflow and Quality of Service elements [94, 95], the OGF/GGF [96] standards based GridSAM project [97] and the GridEcon European Union funded FP6 project [98]. We discuss the issues related to defining components, composing them to produce workflows, and the security and QoS aspects related to these tasks. We consider components to be entities representing encapsulated software capabilities or methods which can be seen as black boxes, with clearly defined interfaces specifying input and output. A single component may represent a complex software task, or a much simpler software method or process. The latter is more common and thus more true to our intended model of components. By composing components into workflows, it is possible to specify complex compositions of software methods and processes in order to achieve advanced software tasks. Alternatively these compositions can themselves be encapsulated into components that are exposed for use within other compositions.

We present an overview of our work on the complete component lifecycle, from the initial definition of a component, specifying its capabilities, features and interfaces, through component composition and the generation of workflows, to the execution of workflows and the components on which they are built [99]. We pay particular attention to our work on security within workflow execution and the definition of Quality of Service (QoS) for the execution of workflows.

8.1 Information, information, information

Information is a key basic requirement for the definition of components. Where components are concerned, information is extremely important and a vast amount of information goes into the development of a component [91]. Unfortunately in most component models not all of the information which is available is retained within the component meta-data. The information retained will depend on who is responsible for capturing the information and how methodical they are with their approach. It is often easier to record only the minimum information for a component to run within the developers requirements; however this may limit future use or development of the component within other environments.

When developing components, software methods are designed and built and then sealed or encapsulated within a component container. The component developer has a lot of knowledge about the way their code works, its strong points and weak points, in which situations it is likely to perform well, and when it is likely to perform poorly. Likewise, domain experts may have provided input to the design of the software methods in order to make them more efficient in domain specific situations. Components are designed to be made available to external entities that can then take these components and use them without needing to know how the methods within the component are designed or written. The meta-data defining the component should be sufficient to let the component user know if the component is appropriate. Components follow the Write Once, Use Many paradigm. However, the process of encapsulating software methods as components generally results in the loss of vast quantities of information that reside with the component developer and any domain experts involved in the component development. When standard HPC software is executed, the information built up by the software developer and domain experts is combined with other information such as problem space information that the user holds, resource information held by resource providers and Grid structure information held by Grid Managers. In the case of componentised software, where this information is not available, there is likely to be a significant loss of efficiency when the components are executed.

If the developer and domain experts knowledge about the software can be preserved and built into the component, we provide the potential to resolve this information with the resource providers information and Grid managers information, all of which can be represented as meta-data at the various levels of the execution process. This provides the opportunity for much more efficient execution and a variety of advanced execution services that can be developed with the availability of such meta-data. From a user viewpoint, we can use such information to allow much more abstract requirements to be provided by end users and enable a more efficient execution process. We want to capture information about what the user wants to achieve, not how they think it should be done. We want to be able to reason about requirements, not just know the requirements. We want to know about the meaning and behaviour [91] of
components and use this information to enhance the user experience. The workflow engine executing the components needs to have access to this information and to support this, the workflow language describing the interconnection of components within a workflow needs to carry the information. Current workflow languages do not provide support for the inclusion of this additional meta-data, nor do they allow for reasoning about the workflow or adaptation of the workflow in light of this reasoning. Most workflow languages work at the level of how to achieve the end goal as opposed to what needs to be achieved and what tools do we have to achieve this?. Thus we argue that it is necessary to add this higher level coordination process on top of the workflow language [100].

8.2 Workflow Validation and Optimisation

A workflow contains the description of how components are interconnected and the control and data flows between these components. Other meta-data, as outlined above, can also be held about both the context of the workflow and the Grid on which it will run. This allows us to pre-process the workflow before submitting it to the Grid, allowing various levels of validation and optimisation to be performed [92]. Validation reduces the chance of a workflow being submitted which will fail to execute (syntactic validation) or generate semantically incorrect results (semantic validation). At this stage the validation is carried out with reference only to the workflow itself and the component descriptions which may be used to achieve it at the time when this validation is carried out, we have no information about the execution environment. By splitting the process of validation and optimisation into a non-execution environment phase and an execution environment phase allows us to pre-prune the search space prior to the costly process of mapping components to resources. As information at this latter stage is costly to collect and inevitably out of date the ability to reduce the footprint is significant. In order for a workflow to be syntactically valid, all the connections between components must be syntactically valid, that is, the output from a component must connect to a valid input of another component and all compulsory connections must have been made. A component may have dangling connections as long as these are allowable by the components design. Any loops and conditionals must have valid inputs, outputs and structures. The semantic validation of a workflow is a much harder process to perform. This relies heavily on domain specific knowledge. It may for example be perfectly valid to connect two gene sequence tools together through the number and types of their ports, though knowledge of what these tools are actually doing will quickly distinguish that this makes no scientific sense. Here lies one of the examples of where information can be exploited within the system.

In many cases, a workflow can be represented (and achieved) in different ways, yielding the same results. In cases where there are multiple representations of a workflow, each providing equivalent results, one representation may be more efficient than another. Likewise a workflow may contain redundant elements introduced through the composition of nested components. This may be due to the fact that only part of the nested components activity is required for the new activity or due to repeated use of the same workflow structure. By carrying out an automated analysis and optimisation of a workflow description, it is often possible to modify a users initial workflow description into a more efficient but semantically equivalent alternative. The aim of this process is to optimise the runtime execution of the workflow prior to its execution. This optimisation does not involve the use of any resource information and is based purely on the modification of the workflow structure to make it as efficient as possible. This is achieved through potential re-ordering, substitution or addition of components and possible pruning of the workflow structure [92]. In addition to the workflow optimisation process, we can also carry out resource optimisation. The procedure of scheduling a set of components for execution across a set of resources can very quickly become an extremely computationally intensive process as the number of components and available resources increases. If we can prune the resource space prior to beginning the scheduling process, this can dramatically reduce the complexity of the scheduling problem [92]. This resource optimisation process is carried out without reference to a workflow structure, or any live resource information, but can make use of component requirements. For instance, if we consider a group of components that require x86-based hardware to execute, we can immediately remove all SPARC-based resources from the resource pool. This reduces the number of scheduling permutations that need to be considered, hence reducing the time taken to schedule the execution. Similar processes can be carried out for other criteria such as memory, software licences or topology.

All of the above processes are used to reduce the time that the workflow will take when its components are mapped over the actual available resources based on (temporally inaccurate) data on load and status. This not only reduces the time to perform the workflow to resource mapping but also increases the likelihood of successful execution.
8.3 Workflow Pipeline

The processes we have described in the previous section form the early stages of a workflow pipeline that begins with the initial concept of a process, articulated by a user, and ends with the successful execution of the process [101, 99]. The procedure involves the initial abstract concept of connected processes the Conceptual Workflow being reified as a Design Workflow, a valid set of real components interconnected either directly or through workflow constructs such as loops and conditionals, representing a valid workflow [91]. Once a valid, concrete workflow description has been produced, the execution process can begin. The workflow is first mapped onto the execution environment, with components being assigned execution resources, perhaps requiring reservations, resulting in a Planned Workflow. The Planned Workflow represents a state where the workflow is ready to execute and generate results. The Executing Workflow can be observed, and a feedback loop can provide information back to the planning phase in order to carry out either live modification of the Planned Workflow to increase execution efficiency, or to store information for use in optimisation of future workflow planning through the use of a performance repository [102]. We define the three main stages of the workflow pipeline (see Figure 20) as specification, planning and execution, with an optimisation feedback loop between the planning and execution phases. Security services can, and should, exist at all levels of the workflow stack in order to ensure that access to workflow nodes can be controlled and only authorised users can access and request execution of components.

8.4 Component Abstraction

We have previously stated that the aim is for users to provide an abstract description of their requirements and for the workflow execution engine to determine how best to execute the workflow [92, 102]. This applies equally to individual components. If a user specifies that they want to use a Bi-Conjugate Gradient (BiCG) solver implemented in Java, the speed of this solver is purely dependent on the type of resource that is used to execute it. If we can abstract away from specifics and say that we want a Linear Equation Solver, there are opportunities for the workflow engine to select the type of solver most suitable for the available hardware and type of input data [91]. Figure 21 illustrates this with abstractness increasing as you move up the figure. It is, however, not always possible to abstract fully in this manner. For example the user may have selected BiCG specifically as they know it is the best solver for their problem. This information also needs to be captured.

We therefore define components according to an inheritance hierarchy. At the top level is a very abstract description of the component, e.g. Linear Equation Solver. As we move down through the levels of the tree, the component description becomes more specialised. The leaf nodes of the inheritance tree are concrete components for which an implementation is available. For example, in the case of a Linear Equation Solver, the second level of the tree may contain BiCG solver and LU solver. Further specialisation at the next level of the tree may contain a C++, Java and ScaLAPACK implementation for the BiCG solver and C and Java implementations for the LU solver.

A similar approach has been taken in the K-WF Grid project where they have used semantic matching in order to move from a simplified users view of a workflow through to an exactable workflow [103]. This is a similar approach to ours though depends on a more abstract matching between services as opposed to the hierarchical approach used here.
8.5 Workflow Scheduling/Planning

The scheduling process that develops a Planned Workflow from a Design Workflow can be made even more efficient through the addition of performance information [102]. Using a performance repository, a store that takes information about the performance of each workflow execution task, we can add performance information into the scheduling process to allow account to be taken of the performance of resources when executing similar components in the past. This information needs to be augmented, as this information is statistical in nature. Bounds need to be placed onto the accuracy of the information and reservations made for those components which are time-critical to the overall workflow [104].

The notion of the performance repository is a location to hold information collected previously for use in improving future runs. We use two techniques to collect this information. The first, and more simple, is to collect benchmarking information for a resource within the Grid. For example, the process of getting information from a CCD camera for use in an HPC image processing algorithm may be benchmarked and performance information fed into the performance repository to further aid scheduling. This information may be stored as an analytical model derived from the benchmarks or alternatively may be raw data on which interpolation and extrapolation are used to determine estimates for uses not carried out as part of the benchmarking. A more dynamic approach is to use in-line benchmarking where each time the resource is used information is collected as to how the resource functioned with this information being fed back into the performance repository. These two approaches can be used together, which works well as it avoids the bootstrapping problem of a new resource never being selected for use as there is no performance information held about it. Information collected through feedback may later be processed into analytical models and / or pruned to more relevant data. A third approach, which we have yet to evaluate, is the analysis of a service to derive the performance information. For example, with software this could be code analysis, or with hardware an analysis of mechanical time requirements [105]. Truong et al. [106] present an approach for collecting performance information within the ASKALON [107] toolkit. Although we feel that this is a move in the right direction we feel that the addition of a full feedback loop for all components and interpolation and extrapolation would be a benefit to their work as it has been to ours.

The scheduling process is carried out by the Planner and we call the process of scheduling using performance information Performance-Aware Planning. The Planner has a number of potential issues to deal with. Firstly, as the number of components, resources and performance information records increases, optimisation algorithms for handling the planning can become very slow. This is as a result of the vast number of permutations that exist, and the huge number of calculations that need to be carried out to identify the best option (where best is defined by some combination of the users and resource owners requirements). As the number of entities involved in the planning process increases, the number of necessary calculations to identify the best combination increases exponentially. Fur-
ther, obtaining relevant live performance information and resource status from hardware resources can be slow and inevitably leads to information that is out of date by the time it is received. We harness the approaches outlined above to reduce this search space thus making the process of scheduling more tractable. Evaluation of different scheduling algorithms have been adopted, these include: Random, Best of n Random, Game theory [108], Constraint Equations [109], Stochastic Scheduling [110], and capacity planning [111].

8.6 Quality of Service (QoS)

When a user submits a workflow they often have requirements on how the workflow needs to execute. This may be over such factors as how long parts of the workflow take to execute, certain tasks being completed by (or not started before) a given time, the reliability of resources used, or the availability of resources used. This information, often referred to as Quality of Service, needs to be taken into account when planning how a workflow will be executed. QoS can be defined into two main categories based on how important it is to the user that they are achieved: those of Strict (hard) and Loose (soft). In the case of strict reservations the workflow should only be accepted if the planner believes that the workflow can be achieved within the QoS requirements, this in general will lead to a reservation being made for those elements which are strict. On the other hand for loose QoS requirements the user is willing to accept a certain degree of failure to meet the QoS requirements. The user will indicate the proportion of times that the requirement must be achieved and provided that the planner is confident enough that it can succeed in enough cases then the workflow can be accepted [112, 113].

Reservations can be made for the execution of components within a workflow, we refer to this execution process as a task for simplicity. Reservations fall into four main categories. In the first case the user presents a pre-made reservation to the planner, this may be generated from an external source such as a site where the user is informed that their sample will be made available on the test apparatus at a given time. The second case is where the user knows that a reservation will be required on a known resource at a known time. This could be a user who knows they want to use a particular piece of equipment but wants the planner to make the reservation for them. The third case is where the user knows that a reservation will be needed but not where or when. For example the user knows that a telescope will be generating a large number of files for them but is not specifically bothered where the files will reside. In the last case the user does not request a reservation. This does not imply that there wont be a reservation made for this task as the planner may determine that to achieve the overall workflow a reservation needs to be made for the task [112, 113].

In order to apply a users QoS requirements over a workflow document, we have developed the QoS description to be represented as a sister document to a Business Process Execution Language (BPEL) workflow document. This allows us to utilise all the standard BPEL tooling such as editors and workflow engines whilst still allowing easy representation of the QoS elements. The QoS document consists of pointers into the BPEL document using XPath along with QoS requirements over the relevant part of the BPEL document. Alternatively the partner links specified within can be tagged with QoS requirements [114]. The QoS document and the BPEL document flow through the stages of the planner. First a resolver determines if the workflow can be achieved along with determining which resources need reservations based on information from the Component descriptions and the performance repository. The resolver then makes requests to the reservation service for the required reservations. If successful the documents progress to the next stage, if unsuccessful they return to the resolver for re-evaluation. Once the reservations have been made the BPEL document can be submitted to a standard BPEL engine and both documents submitted to the observer. The observer is responsible for monitoring the progress of the workflow through the engine. It should be noted that the workflow may over perform based on the original output from the planner as well as under perform. Both circumstances can lead to problems in the future. If this is going to significantly affect the performance of the workflow then the observer will return the workflow and QoS document to the planner for re-evaluation [114].

8.7 Workflow Security

One of the main drawbacks of using a commodity BPEL engine is in the area of security. The BPEL specification was developed around the orchestration of Web Services to achieve an overall result on the Internet. In this respect there is a significant difference between the Internet and the Grid. In general, services on the Internet can be used by anyone who is authorised to do so. Although this is also true for the Grid the identity of the user is typically more important. A service placed on the Internet may require authentication in order that a user may access it, though if this service invokes other services the authentication is done against the service not the original user. This is largely due to the fact that services on the Internet tend to be more constrained with what they will request of another service, whilst for the
Grid the new invocation could be an arbitrary piece of code. Thus the newly invoked service wishes to know who the original user is rather than the intermediate service which is passing on the request, using this information to determine if the service invocation should be accepted. This is true of the BPEL engines that are currently available. The user may make secure connections to the BPEL engine and the engine will make secure connections to the services that it calls. However, the credentials used by the BPEL engine to talk to other services will be its own and not a proxy of the original submitting user. In order not to break the security model used within the Grid, providing a mechanism which allows all users to use all resources, the users proxy needs to be used for calls from the BPEL engine. This also needed to be achieved without the need to modify the BPEL engine as this would lead to code tracking and fixing for all future releases. Figure 22 illustrates the approach used in this case for the use of X509 certificates. It exploits the fact that we can place helpers into the incoming and outgoing messages for Apache Axis. When a user wishes to use a service they first delegate their proxy to the workflow system. This is stored for future use. When the user submits a workflow their proxy will come in with the submission. Provided that this is already held within the workflows cache of proxies a token can be written into the BPEL document (through the Invoke operation) which can then be copied to all outgoing messages from the BPEL engine. When the messages leave the BPEL engine the token can be used to identify the proxy and the proxy used to sign the outgoing message as expected [112]. This does require modification to the BPEL document though this is not difficult to perform automatically. The same approach has also been used for passing Kerberos tokens through the BPEL engine.

8.8 Related Projects

We conclude this document with a look at some of the projects being carried out at the Imperial College Internet Centre that have been the basis for the development of the work described in this document. ICENI Imperial College e-Science Networked Infrastructure: ICENI has been in development at Imperial for several years (2000-2007). It is an advanced middleware providing facilities for component specification, development, composition and execution. ICENI annotates components with significant component meta-data describing meaning and behaviour of components [115]. Components can be represented as hierarchies with abstract components being built into workflows and the ICENI execution engine selecting suitable concrete component implementations at runtime. In [116], Bubak et al. propose an alternative approach to workflow composition, defining the Application Flow Composer System (AFC System) that takes incomplete workflow composition information and uses knowledge of the Grid environment in order to compose a complete Grid application workflow description.

ICENI provides a graphical workflow composition environment, a variety of schedulers utilising different scheduling algorithms, and a variety of launchers that can launch components on a variety of execution platforms including individual machines and Condor and Globus managed resources. ICENI is based on a pluggable framework and offers an API that makes it possible to develop and plug in new schedulers, launchers and components as required. All entities within ICENI (including hardware resources, components, schedulers, launchers, etc) are represented as services. In addition, ICENI provides advanced role-based access management capabilities that allow each individual entity within the environment to have an attached policy that determines who can access the entity and when they are able to access it. Access control is based on X.509 certificates.

The original version of ICENI was developed on top of the Java Jini service framework. As the ICENI system became larger with more features provided, the platform became increasingly inefficient. In order to rectify this
problem ICENI is being redeveloped on an updated, standards-based framework of loosely coupled Web Services. The various components of the ICENI II system are available as individual packages for use in their own right. When complete the full ICENI II system will provide the functionality and flexibility of the original ICENI implementation, built on an updated and much more efficient Web Service platform. GRIDCC: The GRIDCC project was a European FP6 project that finished in September 2007. Imperial Colleges work in GRIDCC was carried out jointly by the Imperial College Internet Centre within the Department of Computing, and High Energy Physics group within the Department of Physics. The project worked on the development of techniques to manage the realtime control of scientific instruments through a Grid Environment. The Imperial teams developed a workflow environment to manage the composition and execution of GRIDCC workflows. This included a visual workflow composition environment and the security elements of the workflow management system that secure the components within the execution engine ensuring that only allowed entities can execute components and gain remote access to instruments.

GridSAM: GridSAM (Grid job Submission And Monitoring) is a job submission and monitoring framework implemented as a Web Service. GridSAM is one element of the ICENI II platform and is also extensively used in its own right. GridSAM provides a standardised Web Service interface for submission of jobs onto a variety of underlying hardware platforms. A plugin architecture allows easy development of connectors for deployment on other Distributed Resource Managers.
9 Workflows in ProActive/GCM

by N. Ranaldo, E. Zimeo (UoS)

A challenging goal in Grid context is simplifying application development through the composition and integration of functions, such as parametric search applications, data mining routines, simulation systems, etc. Software composition can be achieved by following two different approaches: composition in space and composition in time. In the former approach, static structural relations among units are explicitly modeled. In the latter approach, the dynamic behavior of a system is modeled by defining the temporal relations among actions performed by software components. Adopting a simple analogy with modeling an application through UML diagrams, composition in space can be expressed by class diagrams, in which static relations among components are modeled (association, aggregation, generalization, etc), while composition in time can be expressed by activity and sequence diagrams, in which the temporal order of actions performed by components to deliver a system functionality are modeled. Both the approaches have brought to (1) well-consolidated techniques for designing applications in distributed systems by re-using existing software units and (2) easiness in programming thanks to the separation of functional aspects from non-functional ones. As consequence, application developers are only responsible to specify functional logic of components, while non-functional aspects, such as deployment, load balancing, fault tolerance, security, and concurrency control, are handled by component containers.

Typical component-based systems (such as frameworks based on Fractal component model [76], GridCCM [77] and more recently SCA [78]) follow the composition in space approach and allow to effectively model sub-systems through well-known micro architectures. In this case, well-defined structural relations among components lead to tightly-coupled systems, which are acceptable if a "closed world" assumption is considered for the solution domain. However, since no information on control and data flow is available with this kind of composition, performance enhancing mechanisms, such as efficient scheduling or data management based on locality principle, are difficult to apply. These limitations are driving new research efforts that aim at introducing behavioural descriptions of the system in space-based compositions, such as in ICENI [79] and in GCM [80].

On a different trend, new technologies based on large-scale systems, such as the Web, are promoting a new vision of software assembling and knowledge aggregation where the previous assumption of closed world is overcome by knowledge reasoning in an "open world" [81]. In this new scenario, knowledge is partial and evolving. Therefore, tightly-coupled applications are not ideal for tackling the rapid evolution of knowledge and the increasing number of software components available in the Web. To solve open world problems, service-oriented systems have shown to be more suitable than component-based ones, since through the complete separation among clients and service providers through service registries and mediators, such systems allow to dynamically build loosely-coupled applications, by the composition of services completely independent from each other and separately implemented adopting different technologies. Such loosely-coupled applications are typically described adopting an high-level language for modeling the data and control flow (workflow description) among functionalities required to solve a problem and are managed in service-oriented architectures by Workflow Enactment Systems (WESs) [82]. In our opinion, in Grid programming, in which performance and scalability issues are to tackled, both advantages of component-based architectures and service-oriented ones could be exploited. Moreover, since Grids are characterized by dynamically variable and heterogeneous resources, composition needs sophisticated mechanisms to accomplish two against forces: easy application modeling and fulfillment of quality of service (QoS) constraints. To this aim abstract modeling of workflows should be supported by WESs in order to write Grid applications independent from deployment details and different QoS specifications.

In this contribution, we present the adoption of workflow programming and workflow-based technologies for the enactment of applications composed of function delivered by ProActive objects [83]. It is a preliminary work towards the effective development and efficient deployment of temporarily composed loosely-coupled applications made of GCM components [80]. ProActive objects, in fact, can be seen as simple and "atomic components" whose non functional aspects, such as life-cycle, request management and deployment are managed by the framework, and are the basic building blocks for the development of the GCM-compliant implementation based on ProActive. SAWE [84] was adopted as workflow engine for its features of flexibility and extensibility, and the ProActive Scheduler [85] was integrated in order to achieve dynamic binding of ProActive objects on a pool of managed resources.
9.1 Contribution

9.1.1 ProActive/GCM and the ProActive Scheduler

GCM (Grid Component Model) [80] is a component model for Grid computing whose specification is currently being defined within the Institute on Programming Model of the CoreGRID Network of Excellence. It is an extension of the Fractal component model [76], a simple and extensible model that enforces a strict separation between functional and non-functional aspects, hierarchical composition and collective communication mechanisms. GCM features are defined as an extension of the Fractal specification in order to better target Grid context, and are related in particular to parallel computation, interoperability, deployment, dinamicity and autonomous behaviour. In CoreGRID community, the interoperability with service-oriented systems is faced proposing a Web Service interface for components. Moreover, the necessity to enrich spatial composition with temporal relations has been recently taken into account. A possible solution based on a dynamic scripting language like Ruby [86] is proposed in [87]. It aims at exploiting programming language constructs to easily define spatial and temporal relations and to rapidly prototype and experiment with them. Another solution for spatial-temporal composition is proposed in [88]. The idea is to extend the concept of port for client and server relations to input and output ports in order to model the data control flow, and to use the AGWL workflow language to model the control flow [89]. A reference implementation of the GCM specification is currently under development in the context of the GridCOMP European Project [64], and is based on ProActive. Such work, at the same time, will contribute to the assessment of GCM specification.

ProActive is a flexible and extensible programming and deployment framework that supports Grid applications made of parallel and distributed components, also featuring mobility, security and interoperability mechanisms. Configuration of protocols used to export and access remote objects, also called active objects or ProActive objects is supported. The default version of ProActive uses Java RMI as a portable communication protocol, but other protocols are supported, such as IBIS, JINI and Globus. Non functional aspects of active objects, such as location transparency, activity transparency and synchronization are managed by the framework. Deployment information is separated by functional code through a deployment descriptor, on which the GCM framework is strictly based. ProActive delivers efficiency through asynchronous remote method invocations based on the automatic and transparent future object mechanism.

Recently a Scheduler built on top of ProActive, has been developed and released with the version 3.9 of ProActive distribution [17]. The ProActive Scheduler has the task to dynamically schedule ProActive-based computations of multiple users adopting a specific scheduling policy (FIFO policy is the default one) and delivering additional services of monitoring, fault tolerance, etc. It retrieves available resources interacting with a ProActive Resource Manager, responsible to manage and deliver information about a pool of resources. An execution request to the ProActive Scheduler contains only functional description of the application and is independent from deployment details to access and interact with computational resources.

9.1.2 SAWE

SAWE (Semantic and Autonomic Workflow Engine) is a WES compliant with the WfMC abstract reference model [90], whose main objective is to deliver advanced functionalities for creating, managing and enacting workflow processes, in business, engineering and scientific domains.

SAWE engine is based on a three-layer architecture (see Figure 23). The Control Layer receives the process description in XPDL (a translator from WS-BPEL to XPDL is also available), creates and navigates the process graph (a set of nodes and links that represent the active instances of the process) and chooses the activities that could be executed according to the activation conditions and the control flow of the executing process. Activities can be described following concrete and abstract modeling mechanisms.

The Binding Layer is mainly responsible for associating a concrete resource to each abstract activity in the process. For an abstract activity only functional aspects have to be provided, while no binding information is required. In case of Web Services, the Binding Layer can interact with a Matchmaker that is responsible to perform QoS-based discovery and selection of the most appropriate Web Service instance to be assigned to an activity for the execution by using syntactic or semantic annotations.

The Interaction Layer enables remote invocations to implement interactions with the selected resources. At this level, several communication technologies are supported: RMI, Web Services, POJO, HTTP, etc. The interaction with the resources is demanded to a specific adapter for each technology and is managed using a standard Resource Interface that hides communication technologies to the upper layers. In this work, we focus on using XPDL and
SAWE to build, manage and enact workflows composed of tasks performed by distributed ProActive objects (called ProActive-based activities).

To describe an application workflow composed of ProActive-based activities, we exploited XPDL extensibility and flexibility features to avoid language modification and to grant compatibility with different vendor-specific workflow engine implementations. In particular, we adopted the concept of extended attributes, which are attributes defined by users or vendors to allow for extending XPDL specification according to particular needs.

### 9.1.3 SAWE and ProActive-based Activities

In XPDL the external services involved in a workflow, called applications, are described in the section of application declarations. For each application, some information is to be specified: the application type, formal input and output parameters and eventual extended attributes. The specification provides some applications types for interaction with standard technologies, such as Web Services, EJB, form, business rules and Java classes (called POJO). From the functional viewpoint a ProActive-based activity can be simply modeled as a method execution transparently invoked on a remote object. For this reason, it is possible to describe a ProActive object (also called ProActive application) and its invocation interface (method signature) involved in the workflow definition using a POJO application description. In more details, ProActive applications are defined in a workflow by specifying:

1. POJO as application type;
2. POJO class and method name elements to respectively specify the class of the ProActive object and the method to execute;
3. formal parameters adopted to interact with the invocation interface, that means to describe the method input and output parameters;
4. extended attributes adopted to specify the use of the ProActive middleware to interact with such remote object and information for non-functional requirements, such as security, deployment, etc.

In XPDL specification, the data and control flow is described in the process definition section, which moreover contains the description of each elementary activity involved in the workflow. A ProActive-based activity is characterized by:

1. the ProActive application used to implement it;
2. mapping among formal parameters of the ProActive application to actual parameters, a mechanism used in XPDL to pass values between process and applications.

From the implementation viewpoint, when a workflow is submitted, the Process Control component of the Control Layer retrieves the process definition, navigates the corresponding graph and chooses the ProActive-based activities that can be enacted on the basis of data and control dependencies specified in the workflow description. When an activity has to be enacted, the Activity Control component passes it to the Invoker component of the Binding Level, responsible to identify the communication middleware to enact the activity, in this case the ProActive Resource Interface at the Interaction Layer. This component acts as a communication technology adapter that directly interacts with the ProActive Scheduler. In particular the Invoker interacts with the ProActive Scheduler, which will be responsible to deploy, schedule and monitor the execution of the activity (see Figure 1). The ProActive Scheduler API and information retrieved from the XPDL description are adopted to:

1. dynamically define a job, composed of a task including the execution of the specific method specified in the XDPL file;
2. asynchronously submit it to the Scheduler;
3. retrieve execution result when it is required as actual parameter for the enactment of one or more activities of the application workflow.

We are currently working to the use of ProActive/GCM implementation to build and efficiently enact application workflows based on GCM components. We aim at increasing the binding capabilities of SAWE, allowing the user to specify, in the XPDL workflow description QoS requirements. In particular, since ProActive/GCM components can
be exposed as Web Services, it could be possible to easily exploit the already developed Matchmaker to find and select resources on the basis of functional and QoS requirements. Such aspect, together with the study of mechanisms to associate QoS descriptions to ProActive/GCM components will be taken into account in a future work.

9.2 Positioning in the Institute activities

This contribution concerns the combination of composition in time of service-oriented architectures based on WESs and composition in space of component-based systems. As a consequence this work can be set in the context of integration activities related to the Institute on Programming Model and the Institute on Grid Information, Resource and Workflow Monitoring Service.

The Institute on Programming Model, aiming to deliver a definition of a lightweight, component programming model for development and execution of grid applications, is mainly concerned with spatial composition. In particular the task “Component Definition” aims at a standard definition of the Grid Component Model, a specification of Grid components and their composition. On the other hand, the Institute on Grid Information, Resource and Workflow Monitoring Service aims at the development of general and scalable approaches to an information and monitoring infrastructure for large scale heterogeneous Grids. In this context workflow programming and the development of workflow management systems where complex job workflows could be easily deployed, executed, and monitored are investigated in various tasks.

This contribution proposed the adoption of SAWE for the enactment of workflows composed of ProActive-based activities leveraging a scheduling system for transparent and dynamic deployment and management of ProActive objects. Such work can be useful in the direction of the definition of workflow engines for the effective development and efficient deployment of workflows of GCM components’ functions. In particular a future work is the definition of deployment and scheduling mechanisms, such as mechanisms based on locality principles, and their integration in WESs able to optimize performance and to satisfy QoS parameters.
10 WP3 activities at UoL
by J. Cunha (UoL)

10.1 Contribution

10.2 Positioning in the Institute activities

References


CoreGRID TR-XXX 41


[38] H. Liu and M. Parashar. DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. In 9th Int. Euro-Par Conference (EuroPar 2003), LNCS.


[66] Institute, P.M.: Basic features of the grid component model. CoreGRID Deliverable D.PM.04, CoreGRID (march 2007)


CoreGRID TR-XXX


