Proceedings of the Programming Model Institute Technical meeting 2008

M. Aldinucci[a], A. Basso[b], F. Baude[c], A. Bolotov[b], H. Bouziane[c], S. Campa[a], D. Caromel[c], J. Cunha[d], M. Classen[e], P. Classen[e], J. Cohen[f], M. Coppola[a], N. Currle-Linde[g], M. Danelutto[a], P. Dazzi[h], J. Dünnwebber[i], S. Gorlatch[i], L. Henrio[c], P. Kilpatrick[j], C. Lengauer[e], S. Mc Gough[f], C. Perez[c], N. Ranaldo[k],N. Tonellotto[h], E. Zimeo[k], G.Zoppi[a]

CoreGRID Programming Model Institute

Univ. of Pisa[a], Univ. of Westminster[b], INRIA[c], Univ. of Lisbona[d], Univ. of Passau[e], Imperial College[f], HRLS[g], ISTI/CNR[h], Univ. of Muenster[i], Queen's Univ. Belfast[j], Univ. of Sannio[k]

CoreGRID Technical Report
Number TR-138
May 29, 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

Proceedings of the Programming Model Institute Technical meeting 2008

M. Aldinucci[a], A. Basso[b], F. Baude[c], A. Bolotov[b], H. Bouziane[c], S. Campa[a], D. Caromel[c], J. Cunha[d], M. Classen[e], P. Classen[e], J. Cohen[f], M. Coppola[a], N. Currle-Linde[g], M. Danelutto[a], P. Dazzi[h], J. Dünnwebber[i], S. Gorlatch[i], L. Henrio[c], P. Kilpatrick[j], C. Lengauer[e], S. Mc Gough[f], C. Perez[c], N. Ranaldo[k],N. Tonellotto[h], E. Zimeo[k], G.Zoppi[a]

CoreGRID Programming Model Institute

Univ. of Pisa[a], Univ. of Westminster[b], INRIA[c], Univ. of Lisbona[d], Univ. of Passau[e], Imperial College[f], HRLS[g], ISTI/CNR[h], Univ. of Muenster[i], Queen's Univ. Belfast[j], Univ. of Sannio[k]

CoreGRID TR-138

May 29, 2008

Abstract

In January 2008, the Programming model Institute of CoreGRID held a plenary meeting in Paris. The meeting was aimed at discussing progress 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. The meeting was also notable in that it was the first technical meeting where the new Associate partners of CoreGRID (those involved in 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, four contributions are related to autonomic features in GCM (the ones presented in Sec. 4, 6, 7 and 8), and three deal with workflow/component related themes (Sec. 3, 9 and 10). Overall, all the contributions are more or less directly related to GCM and, more generally, 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. Duennweber, S. Gorlatch (WWU Muenster)

This work is an extension of 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 [123, 121] with (2) Higher-Order Components (HOCs [120]), i. e. , components that are readily integrated with the target middleware [119]. 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 [126]. Therefore, we extend the LooPo-HOC by a largely decentralized communication scheme to achieve better scalability [125].

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 of 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*, in which 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).

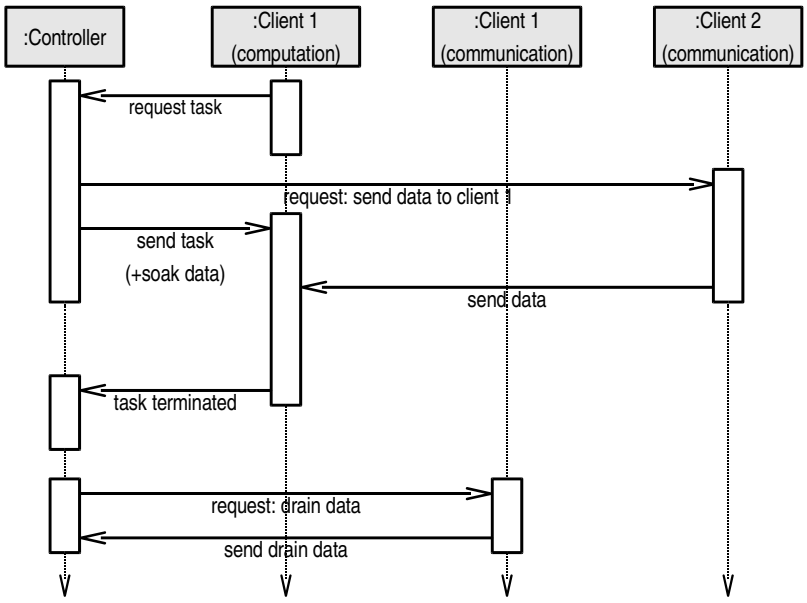

Figure 1: Typical sequence

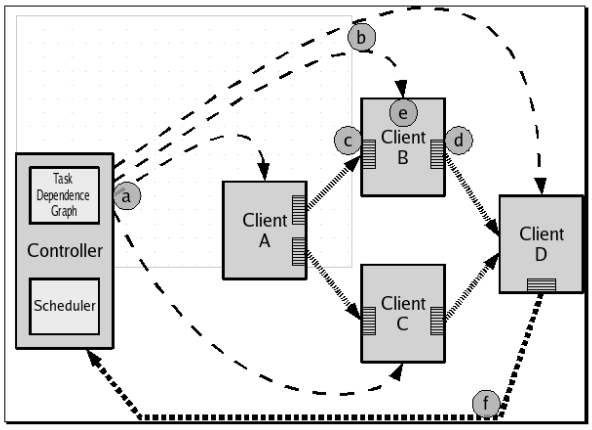

Figure 2: P2P scheme

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 to which the result is being 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 and 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* [124].

We have adapted the classic *tiling* technique for dynamic task farming [119]: 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) [122].

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 [122]) 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 is used for workload monitoring.

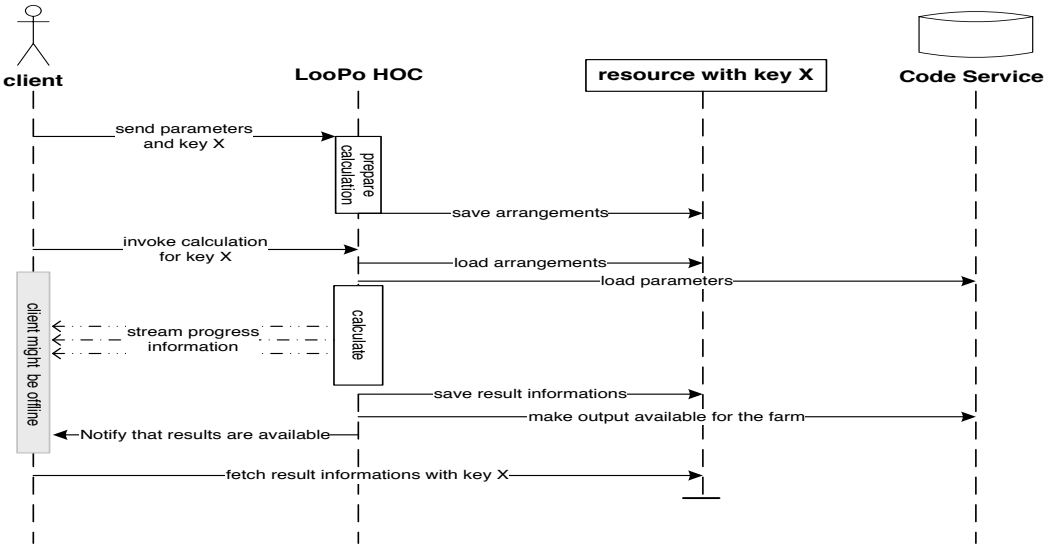

Figure 3: Sequence Diagram for using the LooPo-HOC

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 problem sizes of matrix multiplication and SOR could not be made 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.

1.4 Positioning in the Institute Activities

This work is closely related to the institute's activities, since it deals directly with component-based Grid programming. A Web service is used for accessing the LooPo-HOC remotely, such that it can be connected easily , e. g. , to any GCM application which requires an efficient implementation for running parallel loops on multiple Grid hosts. The suggested combination of components with loop parallelization is also useful in the context of other popular component models (e. g. , CCA [116] or CCM [69]). Aside from the HOC-SA code transfer mechanism, no special features of this or any other particular component technology are required.

The experiments with the LooPo-HOC have shown that the integration of HOCs and GCM components with other tools for distributed programming can further simplify the development of Grid applications and increase their performance at the same time. Beside LooPo and HOCs, GCM components can be integrated easily, e. g. , with the KOALA scheduler [118] for a user-transparent, automated choice of execution machines [117] (as studied in CoreGRID WP 6).

While scheduling GCM applications with KOALA increases their performance by an automatic adaptation of execution environment (i. e. , switching between resources), the integration of GCM with HOCs and LooPo helps to reduce execution times by an automatic adaptation of the application code.

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) [47] 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 [47], 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 [47].

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

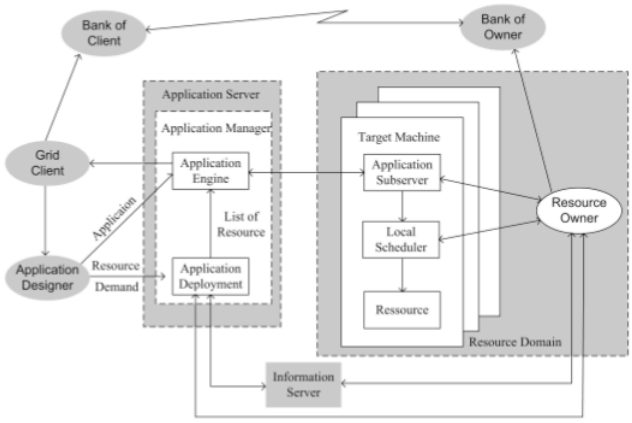

Figure 6: Grid organization within an economical scheme of distribution of resources

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[63]. Spatial composition principle is followed by most component models like GCM [47], FRACTAL [70], CCA [64], CCM [69], 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[63]. There exist many environments that deal with temporal compositions such as workflow systems like ASKALON [62], TRIANA [66], KEPLER [67], BPEL4WS [68] or other cited in a taxonomy of Grid workflow systems provided in [65].

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.

Required concept	Provided concepts	Selected strategy
Task-Component	provided, used operations and tasks	extend GCM with task concept
Ports	spatial: GCM ports temporal: input and output data	extend GCM with temporal ports
Composition	spatial: GCM bindings temporal: data and control flow from AGWL	extend AGWL with GCM components and spatial bindings
Life-cycle	states and transitions	inferred from composition

Figure 7: GCM and AGWL concepts used for defining a spatio-temporal model.

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 [62]. 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
`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.

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

```
1 <component name="name" (extends="...")?>
2 <dataIn name="name" (type="...")?
3 set="outputRefORvalue"/>*
4 <clientPort name="name" (type="...")?
5 set="serverRef"/>*
6 ...
7 <attribute name="name" (type="...")?
8 set="attrRef">*
9 <body>
10 ( <component>+
11 <setPort client="name"
12 server="nameOrData"/>?
13 )*
14 </body>
15 </component>
```

Figure 9: Main spatial composition elements.

```
1 <sequence name="name">
2 <dataIn name="name" type="..." (set=..)?/>*
3 <dataOut name="name" type="..."/>*
4 <clientPort name="name" type="..." (set=..)?/>*
5 <serverPort name="name" type="..."/>*
6 <!-- other spatial ports -->
7 <component>+
8 </sequence>
9 <if name="name">
10 <!-- ports like in sequence-->
11 <component>*
12 <condition> condition </condition>
13 <then> <component>+ </then>
14 <else> <component>+ </else>?
15 </if>
```

Figure 10: Control structure examples.

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 actvities

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. Caromel, 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 ground 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 litterature (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 reconfigurations 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

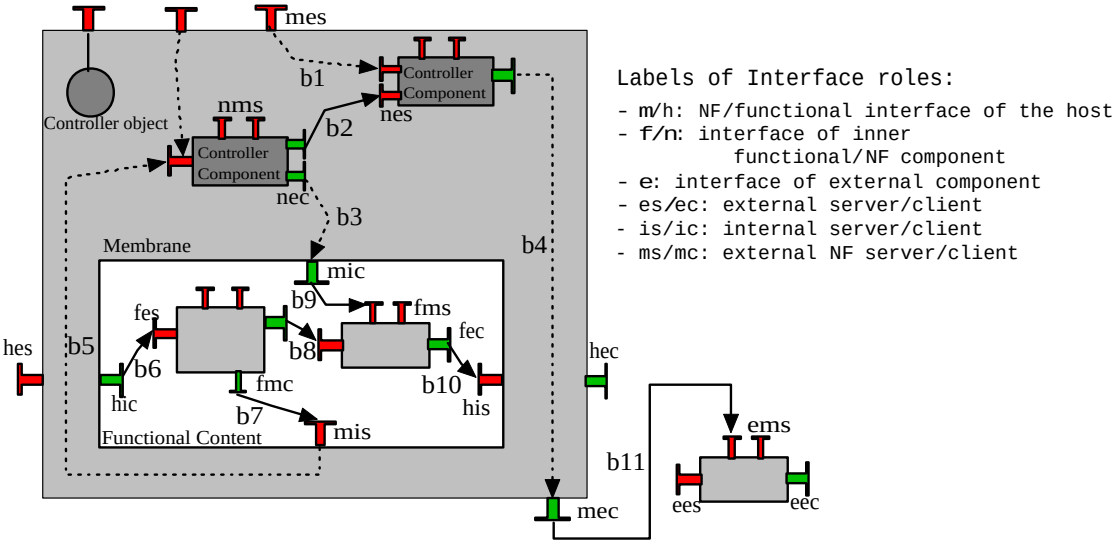

Figure 11: Structure for the membrane of Fractal/GCM components

[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 and A. Basso (UOW)

5.1 Contribution

5.1.1 Introduction

Component models enable modular design of software applications that can be easily reused and combined, ensuring greater reliability. This is important in distributed systems where asynchronous components must be taken into consideration, especially when there is need for reliable dynamic reconfiguration. In these models, components interact together by being bound through interfaces, however, there is a further need for a method which ensures correct composition and behaviour of components and their interaction with the environment.

The recent development of a Grid Integrated Development Environment (GIDE) based on the GCM specification [128] opens new possibilities for the dynamic reconfiguration scenario in large distributed systems. We are able to take advantage of pre-built components in the GIDE (namely the component's hierarchical composition, their API, and the monitoring of both components and resources) to form a basis for a reconfiguration framework which exploits the underlying properties of the specification language and deductive reasoning verification methods used in our research. We consider the monitoring specification of [130] and the state information that can be retrieved through calls to the `LifeCycleController` interface (`getFcState` operation) for components, as well as other monitoring techniques for the environment.

In this work we propose the technique to check a request for a system reconfiguration against these data about the environment of the system. Namely, we show how the data can be formally presented in the framework of so called Temporal Deontic Specification and then verified based on the resolution procedure.

5.1.2 Reconfiguration

We refer to reconfiguration as to the process through which a system halts operation under its current source specification and begins operation under a different target specification [131], and more precisely, after the deployment has taken place (dynamic reconfiguration). Some examples include the replacement of a software component by the user, or an automated healing process activated by the system itself. In either of these cases we consider the dynamic reconfiguration process as an unforeseen action at development time (known as ad-hoc reconfiguration [129]). When the system is deployed, the verification process should run continuously and the system will report back the current states for model mapping; if a reconfiguration procedure is requested or inconsistency detected, the healing process is triggered. The dynamic reconfiguration process works in a recursive way, constantly checking for update requests to the model and taking actions accordingly, enabling us to achieve an automated runtime reconfiguration through cycling deductive verification. The approach here is to specify general invariants for the infrastructure and to accept any change to the system, as long as these invariants hold. We assume that the infrastructure has some pre-defined set of norms which define the constraints for the system, in order to ensure system safety, mission success, or other crucial system properties which are critical to the system.

5.1.3 General Method

The following are the specification requirements that set up the configuration framework:

- Formal specification of component system
- Formal specification of the infrastructure
- Custom extension of specification language to allow for efficient re-configuration
- Continuous verification - check for system integrity / re-configuration requests
- Automatic safe re-configuration

The we define the set of the following Specification Constraints:

- Specification for primitive and composite components (Source-code, metadata)

- Specification of safety, fairness, etc.
- Specification of the infrastructure properties.

5.1.4 Temporal Deontic Specification

As a formal framework to analyse configuration we are considering a **[Deontic Temporal Specification]** which is a tuple $\langle In, St, Ev, N, Lit \rangle$ where In is the set of initial constraints, which give us the initial conditions of the system, St is the set of step constraints, i.e. those that indicate possible moves of the system, Ev is the set of eventuality constraints, which represent conditions/requirements of the system that must eventually occur N, is a set of normative expressions, which indicate obligations, permissions and prohibitions on the behaviour of components, and finally, Lit is the set of literal constraints, i.e. formulae that are globally true.

The TDS is a normative extension of the normal used for the resolution based verification defined for CTL like branching-time logics [127]. It uses the temporal logic operations $\Box$ – 'always in the future', $\diamondsuit$ – 'at sometime in the future', $\bigcirc$ – 'at the next moment in time', $\mathcal{U}$ – 'until', and $\mathcal{W}$ – 'unless' together with the path quantifiers: **A** – 'for any future path and **E** – 'for some future path.

For the deontic part we assume a set $Ag = \{a, b, c \ldots\}$ of agents (processes), which we associate with deontic modalities $\mathcal{O}_a(\phi)$ read as 'ϕ is obligatory for an agent a' and $\mathcal{P}_a(\phi)$ read as 'ϕ is permitted for an agent a'.

5.1.5 Specifying Reconfiguration in TDS

We assume that we are given a set of specification properties (S_i being the start state and S_j the end state), a set of norms, N, and a set of invariants I. We can define a reconfiguration, R, to be applicable when the following conditions holds:

- R commences when the initial state S_i is not operating anymore and finishes before the last state to be updated, S_j, becomes compliant with the system.
- S_j is the appropriate choice for the target specification at some point during R.
- Time for R is less or equal than the time for the transition from S_i to S_j.
- The transition invariant(s), I, holds during R.
- The norms, N, for S_j are true at the time when R finishes.
- The lifetime of R is bounded by any two occurrences of the same specification.

5.1.6 Example

Let us consider the following conditions for the reconfiguration of a component system.

- r and s components that can be bound to the system.
- q - new composite component, a composition of r and s.
- r always requires s not to be active in any of the next states
- s requires r not to be bound in some possible development of the system.
- q, should not be bound at the next state
- a request for the permission to eventually bind q

In the figure below we summarize the formal specification of the given conditions.

Conditions of the System	Constraints of TDS
Dependency between counterpart components	$\mathbf{A}\,\Box(r \Rightarrow \mathbf{A}\bigcirc s)$ $\mathbf{A}\,\Box(\mathbf{true} \Rightarrow \neg r \vee \mathcal{O}_i \neg q)$ $\mathbf{A}\,\Box(s \Rightarrow \mathbf{E}\bigcirc r)$ $\mathbf{A}\,\Box(\mathbf{true} \Rightarrow \neg s \vee \mathcal{O}_i \neg q)$
A request for the permission to eventually bind q	$\mathbf{start} \Rightarrow x$ $\mathbf{A}\,\Box(x \Rightarrow \mathbf{A}\diamondsuit w)$ $\mathbf{A}\,\Box(\mathbf{true} \Rightarrow \neg w \vee \mathcal{P}_i q)$

Translation into the TDS

Below we give now the translation of the given specification (represented by steps 1-7) into the TDS.

1.	r	$\Rightarrow$	$\mathbf{A}\bigcirc s$	
2.	$\mathbf{true}$	$\Rightarrow$	$\neg r \vee \mathcal{O}_i \neg q$	
3.	s	$\Rightarrow$	$\mathbf{E}\bigcirc r_{\langle f \rangle}$	
4.	$\mathbf{true}$	$\Rightarrow$	$\neg s \vee \mathcal{O}_i \neg q$	
5.	$\mathbf{start}$	$\Rightarrow$	x	
6.	x	$\Rightarrow$	$\mathbf{A}\diamondsuit w$	
7.	$\mathbf{true}$	$\Rightarrow$	$\neg w \vee \mathcal{P}_i q$	
8.	$\mathbf{true}$	$\Rightarrow$	$\neg s \vee \neg w$	$DRES\,4, 7$
9.	r	$\Rightarrow$	$\mathbf{A}\bigcirc \neg w$	$SRES\,2,\ 1, 8$
10.	s	$\Rightarrow$	$\mathbf{A}\bigcirc \neg w$	$from\,8$
11.	$r \vee s$	$\Rightarrow$	$\mathbf{E}\bigcirc \Box \neg w_{\langle f \rangle}$	$1, 3, 9, 10$
12.	x	$\Rightarrow$	$\neg(r \vee s)\,\mathcal{W}\,w$	$TRES,\ 6, 11$
13.	x	$\Rightarrow$	$w \vee \neg(r \vee s)$	$\mathcal{W}\ removal,\ 12$
14.	x	$\Rightarrow$	$w \vee \neg r$	$classical, 13$
15.	x	$\Rightarrow$	$w \vee \neg s$	$classical, 13$
16.	$\mathbf{start}$	$\Rightarrow$	$\neg x \vee w \vee \neg r$	$temporising, 14$
17.	$\mathbf{start}$	$\Rightarrow$	$\neg x \vee w \vee \neg s$	$temporising, 15$
18.	$\mathbf{start}$	$\Rightarrow$	$\neg s \vee \neg w$	$temporising, 8$
19.	$\mathbf{start}$	$\Rightarrow$	$\neg s$	$SRES 1,\ 5, 17, 18$

We can immediately conclude now that s should not be initially active.

5.1.7 Deductive verification

Our extension of the language $\mathrm{SNF}_{\mathrm{CTL}}$ by deontic operators - Temporal Deontic Specification (TDS) enables us to apply the resolution based verification technique extended to cope with the normative dimension.

Here we present two new resolution rules that will be used in our example are given below.

$$\frac{\begin{array}{l}\mathbf{true} \Rightarrow D \vee O_i l \\ \mathbf{true} \Rightarrow D' \vee P_i \neg l\end{array}}{\mathbf{true} \Rightarrow D \vee D'}\ DRES \qquad \frac{\begin{array}{l}A \Rightarrow \mathbf{E}\,\Box \neg l_{\langle \mathrm{f} \rangle} \\ B \Rightarrow \mathbf{A}\diamondsuit l\end{array}}{B \Rightarrow \mathbf{A}(\neg A\,\mathcal{W}\,l)}\ TRES$$

5.1.8 Loop (Invariants) Detection

Our system can identify normative invariants which should be preserved. What is even more important we are able to detect hidden invariants, i.e. those that are not explicitly given in the specification. It can be observed from the example above that our procedure has detected a loop (invariant) in $\neg w$ which is immediately obvious from the set of TDS clauses. Additionally, this loop, in conjunction with clauses 2 and 7 indicates a hidden 'deontic invalriant' property, that s fires the condition $\mathcal{O}_i \neg q$ and w fires the condition $\mathcal{P}_i q$.

Now, if we assume that r is initially active, then we can continue the proof and derive a contradiction as follows:

20.	**start**	$\Rightarrow$	r	$assumption$
21.	**start**	$\Rightarrow$	w	$SRES\,1,\ 1, 16, 20$
22.	**start**	$\Rightarrow$	$\mathcal{P}_i q$	$SRES1,\ 7, 21$
23.	**start**	$\Rightarrow$	$\mathcal{O}_i \neg q$	$SRES1,\ 2, 20$
24.	**start**	$\Rightarrow$	**false**	$DRES,\ 22, 23$

Thus, a request to bring a composite component, q to the system can only be satisfied if r is not active. Otherwise, if r is bound to the system, the request to bind q should be rejected.

5.1.9 Conclusions and Future Work

We have presented a definition for dynamic reconfiguration and a formal way to specify the behaviour of a component model and its infrastructure. The method allows to prevent inconsistency and suggest corrections to the system. If inconsistencies are discovered then the 'healing' process commences: "re-configure" the computation tree model that conforms the protocol. Our future plans are to embed all these features in a prototype plug-in for the GridComp GIDE and test it on other case studies. Additionally, at the architectural level we are aiming at defining an automata-based approach, where the data for the deductive verification described above will be derived from an automaton that represents the behaviour of the environment of the component system.

5.2 Positioning in the Institute activities

The work discussed is related to the activities of the Programming Model Institute, namely to Task 3.3, assisting to the analysis of an Advanced Programming Model considered on the high level of abstraction. High level specification given in the temporal language not only continues our previous work but also, being currently extended by invoking normative constraints, enables to represent complex behaviour properties of a system, and opens new light to the formal analysis of a reconfiguration process.

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] were transferred to the *Grid* scenario and used to program several different kinds of super/meta/skeleton components modelling common Grid programming paradigms [13, 7, 4]. Component embedding of skeletons somehow filled the gap between skeletons and modern software engineering concepts. Component technology allowed skeletons to be embedded in well-know programming paradigms (the components) and to hide irrelevant implementation parameters from 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 same programming pattern that classical, non-component skeleton programming environments usually provide to the application programmers. Skeleton code parameters are provided as components, allowing hierarchical and incremental program development. Component technology simplified somehow code deployment on remote processing elements leveraging onto component framework 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 management features. Behavioural skeletons were thus introduced in 2007 [5, 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 [14] features are implemented.

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 Architecture 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 work provides service application programmers with a very high level programming paradigm that can be easily used to program most of the typical *embarrassingly parallel* Grid/distributed applications 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 [16].

6.1 Contribution

Service Component Architecture [1] provides the user with a programming framework supporting application development 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 technologies 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 that it provides very useful ways to build applications from existing services (which is the same thing you can do using BPEL in other contexts) and allows the service composition to be (re-) used within applications as a primitive component.

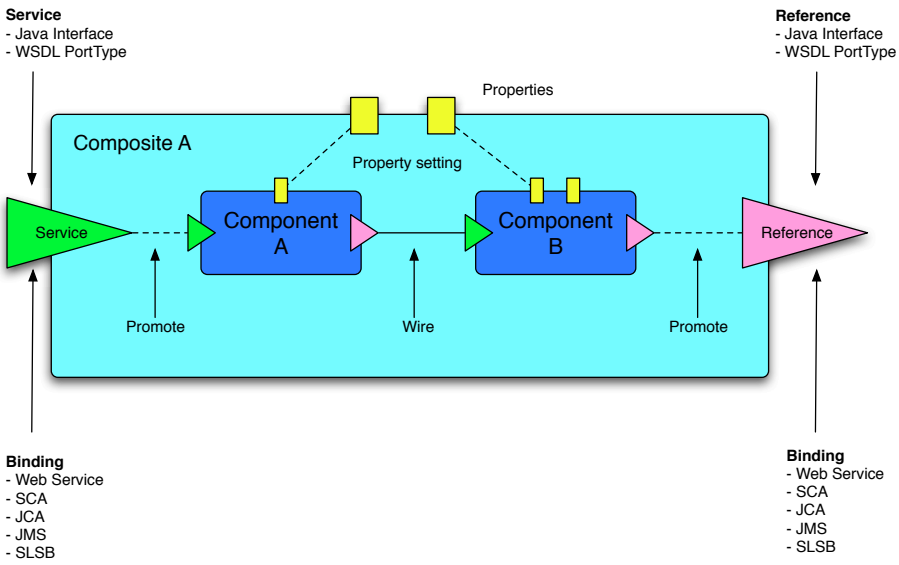

Figure 12: SCA component assembly

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. 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 is 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 requesting from 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 these are functional concerns) as well as to start autonomic management and to submit rules affecting autonomic management behaviour (these are instead non-functional concerns). Autonomic management is implemented using JBoss rules [2]. Each Jboss rule includes a precondition as well as the actions to be executed if the precondition is satisfied. Both preconditions and actions use proper Java beans associated with 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 submitting 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 of the WorkerManagers takes 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-

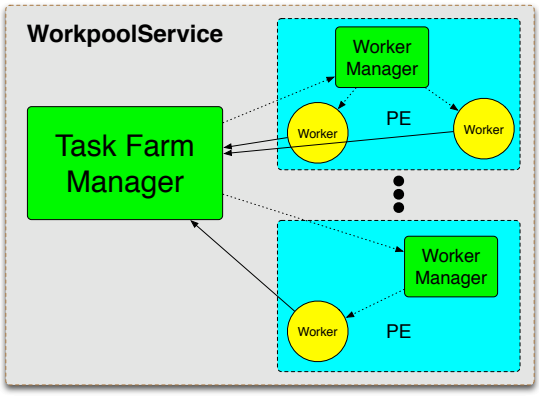

Figure 13: WorkpoolService skeleton

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 deallocate 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 achievement of a service time (time elapsed between 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 tasks (i.e. tasks taking significantly longer (100 times) time to compute than to get the input data and to deliver output results) through 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 appropriate JBoss rule stating that in case the service time goes below a given threshold, new workers should be added, such as

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

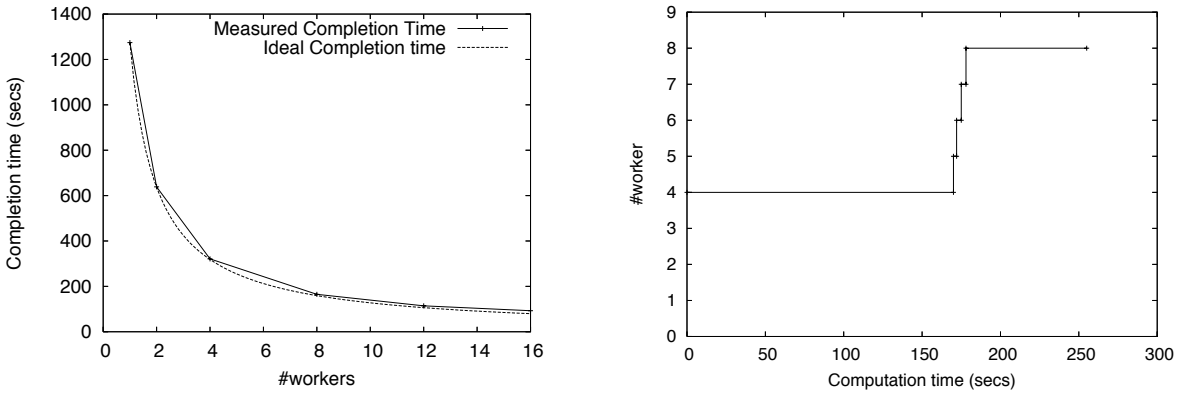

Figure 14: Scalability (left) and effect of dynamic management (right) in task farm behavioural skeleton

After implementing the taks 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 suitable `.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 of implementing programming environments on top of GCM that contribute to raising the level of abstraction presented to 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 employed 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 actvities. 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 Challenges in Hierarchical Autonomic Management

by P. Kilpatrick (QUB), M. Aldinucci, M. Danelutto (UNIPI)

This report sets out the context of our approach to addressing the issue of autonomic behaviour in Grid systems and relates it to earlier ideas on modelling and reasoning about such systems. The work assumes the framework of the Grid Component Model (GCM) [47] and, in particular, the presence in each component of an active autonomic manager. We describe our assumptions in relation to an autonomic cycle and the definition of a contract in the context of a hierarchical system such as that which may be constructed using the GCM.

In general an autonomic system may, inter alia, extend the functionality of the system (for example, in response to some change of the external environment). However, for the sake of simplicity we preclude this possibility and focus on systems where the goal of the autonomic manager is to modify the non-functional behaviour of the system while preserving the functional behaviour. Fundamental to our approach is the following:

1. There should be a clear separation between autonomic management and system functionality.
2. The components should have associated descriptions of functional and non-functional behaviour, preferably described using a recognised modelling language rather than natural language.
3. The autonomic management system should be capable of processing automatically these descriptions of functional and non-functional behaviour.

7.1 Contribution

In earlier work we investigated modelling and reasoning about Grid systems. In particular, we investigated the muskelskeleton-based system [132]. The original version of muskelwas reverse-engineered to produce a model (described using the Orc notation [133]). Examination of the model allowed us to identify clearly the underlying design feature contributing to a recognised (non-functional) weakness, viz. the presence of a single point of failure. By reasoning at the model level we were able to identify and describe a design change which removed this single point of failure while retaining the functional behaviour of the muskelsystem. The model acted as a guide to the modification of the original design and to realisation of the new system. Furthermore, we were able, by reasoning semi-formally at the model level, to argue preservation of the functionality and to make qualitative predictions about performance of the modified version as compared to the original.

The exercise just described represents a manual realisation of the cycle that an autonomic manager must effect. A manager must be able, via monitoring of system activity, to identify the need for change, the nature of change required and trigger realisation of that change. Our aim is to model and eventually implement such a manager. Clearly this is a hugely challenging task. In addressing this task we further narrow our domain of interest to *behavioural skeletons* [5]. A behavioural skeleton may be regarded as comprising a (parameterised) programming skeleton (in the sense of Cole [9]) together with a manager. Thus, our domain is restricted to a well-defined set of parallel/distributed programming patterns which have been found useful. Our belief is that thus restricting the focus will ease the burden of manager modelling.

7.1.1 Autonomic management and hierarchical programming models

A behavioural skeleton can be represented by a tree, whose nodes are graphs of components or primitive components; levels of the tree represents the layers of the system. The edges of the tree represent the implementation/refinement relationship or the "use" relationship. Note that, at least in this general picture, the layers of a system are not necessarily implemented using the same programming model, language or methodology (we are rarely interested in designing all the layers of a system). The GCM component hierarchy covers a number of levels in the top of this tree. An autonomic system is naturally hierarchical [14].

In the framework of the GCM we can also represent the (top of the) tree of nodes which are graphs with a simpler structure. The general structure of a GCM application is sketched in Fig.15 (a); its abstract (functional and management) view is presented in Fig.15 (b) and Fig.15 (c). View (b) represents the structure of the application: dotted lines represent refinement/implementation relationships; solid lines data dependencies (use relationship). For convenience of representation, each composite component has a special child representing itself (striped); in this way all functional

dependencies at a given level remain confined at this level (import/export bindings remain confined within a single sub-tree in order to preserve "membrane intuition").

View (c) is a simplification of view (b) where all functional dependencies have been masked (for simplicity). In accordance with autonomic computing intuition, each (non-ground) node exhibits its own autonomic control. The management of the whole system is achieved through coordination among autonomic managers, in both horizontal and vertical directions. Here, we rely on the assumption of the hierarchical structure to cope with the complexity of distributed management (orchestration and contracts).

Autonomic cycle Each (non-ground) node, $C(i)$, has an autonomic cycle. At the end of each iteration of the cycle the node $C(i)$ knows something about the state of each of its children $C(i+1)_0 \cdots C(i+1)_k$ ($C(i+1)_0$ being its own state, i.e. the striped node). We call these bits of information $um_0, \cdots, um_k$ (for monitoring). Each element of the list is a set of pairs $\langle$feature, value$\rangle$, or the empty set, denoting the timeout $t(i)$ (no state received from a given child for some reason). Each state is marked with an iteration number and is valid for a number $j(i)$ of iterations (indicating the extent of historical information used by node $C(i)$). Overall, $C(i)$ keeps a finite poset (Partially Ordered Set) of states of the children (a tree with depth $j(i)$). Both $j(i)$ and $t(i)$ might be dynamically adjusted (this is part of the autonomic behaviour). With this hypothesis, $C(i)$ may reason with a maximum history that is $O(t(i) * j(i))$ in time (and use $O(k(i)^{j(i)})$ space for keeping children state). Here, all children of $C(i)$ have been managed uniformly, but, if required, it is possible distinguish them further by introducing another index (omitted here, for simplicity).

Overall, we reason on the complete application (tree) by induction. The autonomic cycle of $C(i)$ completes (logically) when the autonomic cycles of all of its children $C(i+1)$ have completed (or at least they have to complete within the $j(i)$ iteration window (a sliding window, since it can be dynamically self-adjusted). In this way, we note that autonomic cycle iteration increases in granularity from the bottom to the top. This matches well the distributed system hypothesis.

Contracts We assume that the behaviour of an autonomic manager is governed by a contract specifying the limits of the non-functional behaviour of that manager's node. Each node (manager) has associated:

- a set of "nf-use" features $um_0, \cdots, um_k$, which represent the state of (the non-functional characteristics of) the children; each um_k may be a list.
- an "nf-provide" feature pm, which represents the state that this node exposes to its parent. Again, it may be a list.
- an analysis function $pm = AF(um_0, \cdots, um_k)$. Since pm is a list, this function may be described as an array of functions. Since um_0 represents the node state, pm may include information about the component membrane (or, in general, its internal state), and historic information (such as how many times this function has been evaluated).
- an evaluation of the analysis function described as a set of constraints, possibly with existential and universal quantification. For example, $pm = [pm_0, \ldots, pm_m] : E(pm) = (pm_0 < pm_1) \wedge (\exists x \in pm : x < pm_m)$. This analysis function determines whether the contract is valid or broken. The quantification helps to describe max, min and other complex formulae.

The information above should be sufficient to capture the idea of contract validity, and in the case of a broken contract, the monitor information that should be passed to the parent. Here it is necessary to distinguish between active and passive autonomicity. In the former case, pm is proactively passed to the parent node (manager) if the contract is broken, or perhaps periodically. In the latter case, the pm are requested via polling, and since there is no manager (it is passive), in general $k = 0, pm = [um_0]$. To complete the analysis phase the manager should be able to understand what is wrong if E evaluates to false. This means that it should solve a SAT problem for E. Consider two cases:

- There does not exist an assignment of pm that makes E true. This means that this component cannot do anything to bring the contract back into the range of validity and the only possible action is to escalate the problem to the parent.
- It is possible to find an assignment of pm that makes E true. That is, the component can solve the problem. It is then a matter of determining how this might be achieved. Let us suppose that $E_q(pm)$ gives a quantification of the quality of the solution. Then the task is to find the optimum value of $E_q(pm)$.

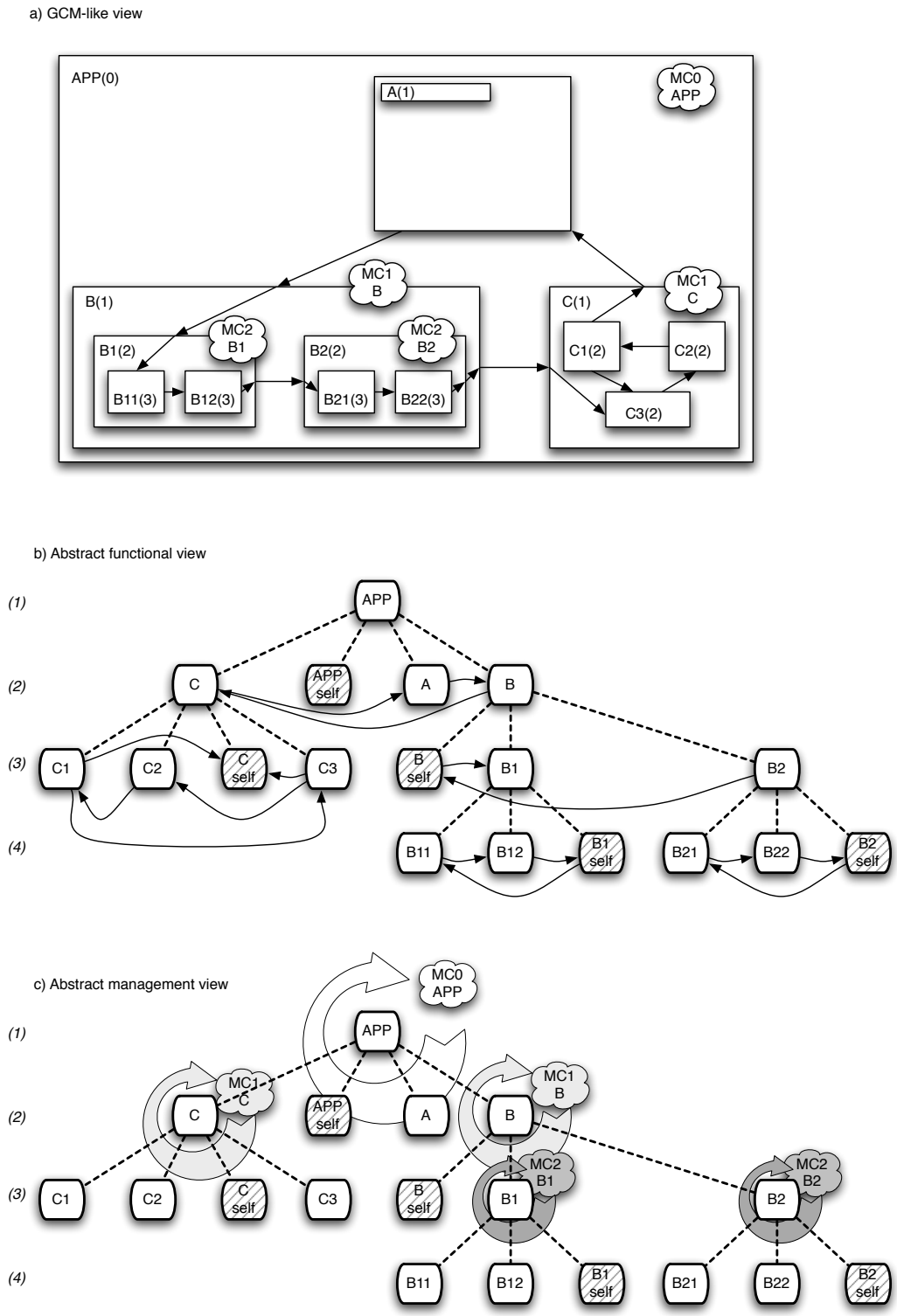

Figure 15: GCM apps

In this process a number of activities are really heuristic. The quantification of the quality of a solution is clearly heuristic, the search space may be limited via suboptimal search, etc. This search should be driven by reconfiguration rules and the forecast of their effect.

Contract design, soundness and completeness We can imagine two kinds of system in which contracts have different roles:

- *Minimal*. The contract includes the function E. The planning activities, the um variables and so on are already defined and embedded in the component. The matching between the information the component needs from the children and provides to the parent is a *design time* activity.
- *Maximal*. The contract carries almost everything. The contract is automatically split and distributed to children. Each node matches what it can do with what it is requested to do. This is much more difficult to achieve and will not be considered in the short/medium term.

In both cases, the role of the behavioural skeleton is to provide a set of strategies that facilitate the determination of values of monitor variables. In the second case, the component should also understand if the pre-defined strategies are suitable to achieve the requested goal. The orchestration of management (or a set of them) is in any case probably embedded in the skeleton, since the strategies depend on it.

7.2 Positioning in the Institute Activities

The work proposed is aimed at describing the interaction of a hierarchy of component autonomic managers in the context of the Grid Component Model and so lies at the heart of the activities of the Programming Model Institute. The CoreGRID WP3 RoadMap: section 3.3 refers to the need for the development of "'Theoretical' computation models that can be used to support the Grid component model, that is abstract programming models raising the level of abstraction provided to Grid programmers." Such a model will transcend particular implementation strategies and will thus be applicable to the realisation of GCM-based applications in areas such as Service Component Architectures (see, for example, Section 6).

8 Advances in GCM

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

8.1 Contribution

Typical Grid architectures are subject to dynamic changes that impact their behaviour [58]. 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 [51]. In recent years, several research initiatives exploiting component technology [55] 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 [61, 51], AutoMate [60], SAFRAN [56], K-Components [57] and finally the forthcoming CoreGRID Component Model (GCM) [55] 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 [2] 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[3].

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 [14]. 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* [5].

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 [54]. 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 [13]). 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-

[2] `http://gridcomp.ercim.org`
[3] `http://www-sop.inria.fr/oasis/ProActive`

mentation.

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.

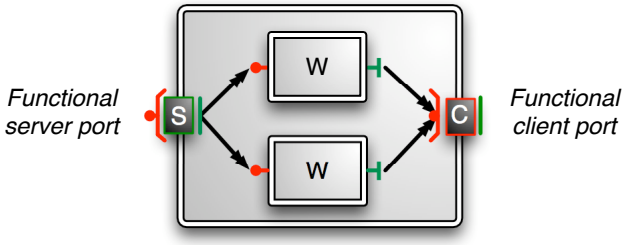

Figure 16: Functional replication behavioural skeleton schema

In [5] 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 16). 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.

8.2 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 $W_1, \cdots, W_n$, and then issues a `stop` on them.

2. *Type Inspection.* All the $W_1, \cdots, W_n$ have the same type. The ABC retrieves from the CC the list of inner components $W_1, \cdots, W_n$, then retrieves TypeOf(W_1).

3. *New.* One or more new inner components of type TypeOf(W_1) are created.

4. *Bind.* The component server interface S is wired to newly created $W_n, \cdots, W_{n+k}$ inner components via the *Binding Controller* (BC). $W_n, \cdots, W_{n+k}$, 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 GridCOMP STREP project [16]. 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.

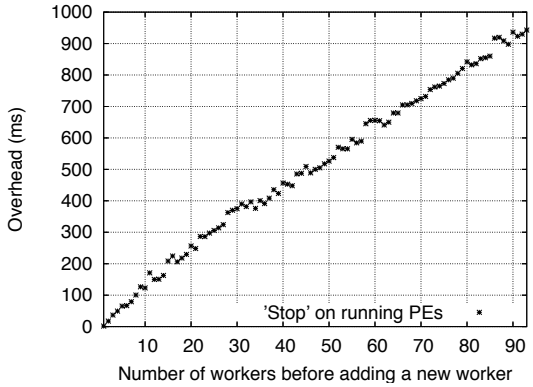

Figure 17: Reconfiguration overhead: Stop.

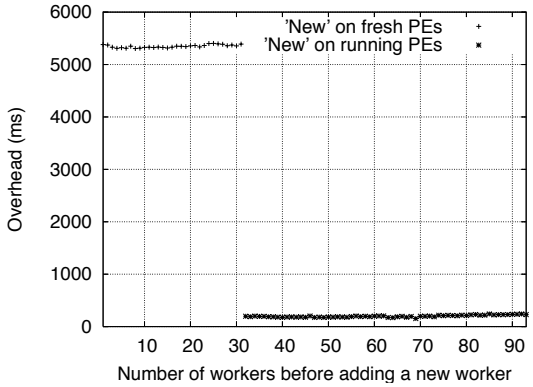

Figure 18: Reconfiguration overhead: New.

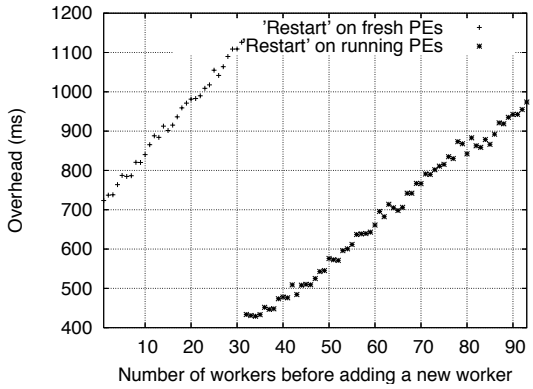

Figure 19: Reconfiguration overhead: Restart.

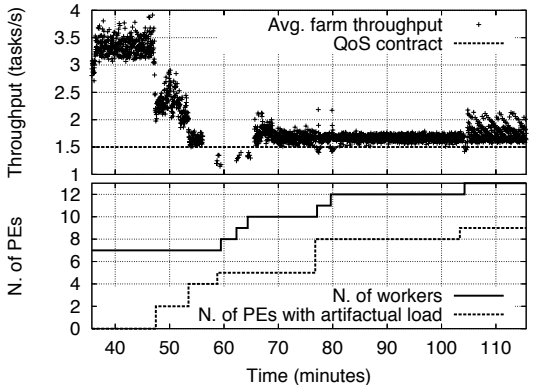

Figure 20: Self-optimisation experiment.

Figures 17, 18, and 19 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. 17 and 19 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 18 shows the time spent for the *new* ABC operation (see Sec. 8.2). 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. 18 and 19 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. 20. 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 (~ 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.

8.3 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.

9 Secure component composition and definition with QoS

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

9.1 Contribution

The London e-Science Centre at Imperial College London has a long history working on Grid Computing and software component related research. Since 2000 the centre has undertaken various projects including the Imperial College E-Science Networked Infrastructure (ICENI) [85, 86, 87], the European Union (EU) funded GRIDCC project (specifically workflow and Quality of Service) [88, 89], the OGF/GGF [90] standards based GridSAM project [91] and the EU funded project GridEcon [92]. Through these projects, and ongoing research, many issues relating to Grids and component technologies have been investigated. Here we discuss elements related to component definition, composition, and the security along with QoS aspects. 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 (as workflows), to the execution of workflows and the components on which they are built [93].

We consider components to be entities representing encapsulated software capabilities or methods, which can be seen as black boxes, with clearly defined interfaces (input and output). A single component may represent a complex software task, or a much simpler software method or process – which is more common and thus more our intended model. By composing components into workflows, it is possible to specify complex compositions of software methods and processes to achieve advanced software tasks. Alternatively, in common with the hierarchical nature of the GCM, these compositions can be encapsulated into new components that are used within other compositions.

9.2 Information, information, information

Information is a key basic requirement for the definition of components, and a vast amount of information goes into the development of a component [85]. Unfortunately in most component models not all of this available information is retained in the component meta-data. The meta-data defining a component should be sufficient to let the external entities using the component know if the component is appropriate. However, the process of encapsulating software methods as components generally results in the loss 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 user problem space information, resource provider information and Grid Manager information. In the case of componentised software, where this information is not retained, there is likely to be a significant loss of efficiency when executing components. If the developer and domain expert's knowledge can be preserved and built into the component, we provide the potential to utilise this information along with resource provider's and Grid manager's information, all of which can be represented as meta-data throughout the system.

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 seek to capture information about what the user wants to achieve, not how they think it should be done; to be able to reason about requirements, not just know the requirements; to know about the meaning and behaviour [85] of components and use this information to enhance the user experience. The workflow engine executing the components needs to have access to this information, the workflow language describing the interconnection of components 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 [94].

9.3 Workflow Validation and Optimisation

A workflow contains a description of how components are interconnected and what control and data flows between these components. Other meta-data, as outlined above, can also be held about both the workflow and the Grid on which it will run. This allows us to pre-process the workflow before submitting it, allowing various levels of validation and optimisation to be performed [86]. Validation reduces the chance of a workflow being submitted which will fail to execute (syntactic validation) or generate semantically incorrect results (semantic validation). However, at the time

when this validation is carried out, we have no information about the execution environment. By performing this pre-execution environment validation and optimisation we can 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. 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. The semantic validation of a workflow is a much harder process to perform and 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 functional knowledge of these tools will quickly show that this makes no scientific sense. This illustrates how the information collected can be used.

Workflows may not represent the most efficent way to achieve a desired goal as they may be the composition of nested workflows which contain redundant elements or a workflow may not achieve a goal in the least computationally expensive manner. By carrying out automated analysis and optimisation of a workflow description, it is often possible to modify a user's 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 more efficient. This is achieved through potential re-ordering, substitution or addition of components and possible pruning of the workflow structure [86]. In addition to the workflow optimisation process, we can also carry out resource selection optimisation. The procedure of scheduling a set of components for execution across a set of resources can quickly become a computationally intensive process as the number of components and resources increase. If we can prune the resource space prior to the scheduling process this can dramatically reduce the complexity of the scheduling problem [86]. 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 permutations that need to be considered, hence reducing the time taken to schedule. 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 it's components are mapped over the available resources based on (temporally inaccurate) data. This not only reduces the time to perform the workflow to resource mapping but also increases the likelihood of successful execution.

9.4 Workflow Pipeline

The processes we have described in the previous sections 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 [95, 93]. 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 [85]. 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 workflow to increase efficiency, or to gather information for optimising future workflows – through the use of a performance repository [96]. We define the three main stages of the workflow pipeline (see Figure 21) as specification, planning and execution, with an optimisation feedback loop between the planning and execution phases. Security services 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.

9.5 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 [86, 96]. This applies equally to individual components. If a user specifies that they want to use a Bi-Conjugate Gradient (BiCG) solver implemented in Java, this significantly reduces the scope for optimisation. If we can abstract away from specifics – Linear Equation Solver – this increases the ability to select a solver most suitable for the available hardware and type of input data [85] – such as a parallel BiCG solver. Figure 22 illustrates this with abstractness increasing as you move up the figure. It is, however,

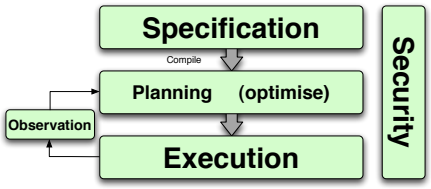

Figure 21: The Workflow Pipeline

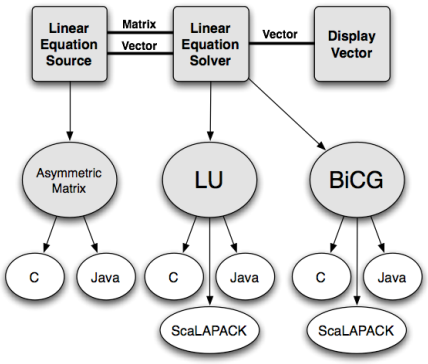

Figure 22: Abstraction of a Component

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, such information 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.

A similar approach has been taken in the K-WF Grid project where they use semantic matching in order to move from a simplified users view of a workflow through to an exactable workflow [97]. This is a similar approach to ours though depends on a more abstract matching between services as opposed to the hierarchical approach used here.

9.6 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 [96]. Using a performance repository, which holds information about the performance of all component, the scheduling process can estimate the execution time for the workflow, or any sub-part of it. Thus it is possible to compare execution times for different component implementations. This information needs to be augmented, as this information is statistical in nature and based on the results from previous executions of the components. 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 [98].

We use two techniques to collect this information. The first, and more simple, is to collect benchmarking information for a resource within the Grid. The second more dynamic approach is to use 'in-line' benchmarking where each time the component is used information is collected as to how the component functioned, feeding this new information back to the performance repository. These two approaches can be used together, avoiding the bootstrapping problem of a new resource not being selected for use as it has no performance information. 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 [99]. Truong et al. [100] present an approach for collecting performance information within the ASKALON [101] 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.

Performance-Aware Planning – scheduling using performance information – has a number of performance issues which are shared with non-performance aware schedulers. This is an NP-complete problem space due to the number of potential resources, component implementations along with the problem of determining the 'best' workflow (where 'best' is defined by some combination of the user's and resource owners requirements) and the performance information for each component / resource combination. The fact that resource status information is slow to collect and inevitably out of date once collected. We harness the approaches outlined above to reduce this search space thus making the process of scheduling more tractable. Evaluation of different scheduling algorithms such as: Random, Best of n Random, Game theory [102], Constraint Equations [103], Stochastic Scheduling [104], and capacity planning [105].

9.7 Quality of Service (QoS)

When a user submits a workflow they often have requirements on how the workflow needs to execute. For such factors as how long (parts of) the workflow takes to execute, tasks being completed by (or not started before) a given time, the reliability or availability of resources. This information, often referred to as Quality of Service, needs to be taken into account when planning workflow execution. QoS can be defined into two main categories based on how significant violation is to the user – those of Strict (hard) and Loose (soft). In the case of strict the workflow should only be accepted if the planner believes that the workflow can be achieved within the QoS requirements. 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 [106, 107].

Reservations can be made for the execution of components within a workflow and 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's sample will be 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 – a user who knows what they want but wants the planner to make the reservation. The third case is where the user knows that a reservation will be needed but not where or when – a large ammount of storage space is required for file though the location is not specific. In the last case the user does not request a reservation. This does not imply that there won't be a reservation made as the planner may determine that to achieve the overall workflow a reservation needs to be made [106, 107].

In order to apply a user's QoS requirements over a workflow document, we have developed the QoS description as a sister document to a Business Process Execution Language (BPEL). 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 or the BPEL partner links along with QoS requirements over the relevant part of the BPEL document [108]. 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. The reserver then makes requests to the reservation service for the required reservations. If un-successful control is passed back to the resolver for re-evaluation. Once the reservations have been made the BPEL document can be submitted to a BPEL engine and both documents submitted to the observer which monitors the progress of the workflow. It should be noted that the workflow may over perform based on the original output from the planner as well as under perform – both can be problematic. If this is going to significantly affect the performance of the workflow then the observer will return to the planner for re-evaluation [108].

9.8 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 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 authenticated to do so though little emphasis is placed on authorisation, while for the Grid, authorisation is as important. Thus on the Internet a service acting on behalf of a user can use its own credentials, this does not hold for the Grid. Thus we need to delegate the users credentials to the invoked service. This is a limitation of the currently available BPEL engines. 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 those of the 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 user's 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.

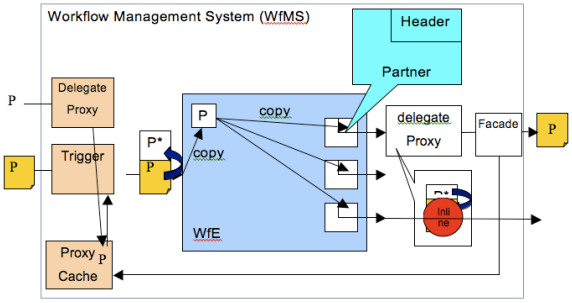

Figure 23: Security delegation through BPEL

Figure 23 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, which it stores for for future use. When the user submits a workflow their proxy will arrive with the submission. Provided that this is already held within the cache 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 engine the token is used to retrieve the proxy which signs the outgoing message [106]. This does require modification to the BPEL document though this can easily be performed automatically. The same approach has also been used for passing Kerberos tokens through the BPEL engine.

9.9 Related Projects

We conclude with a look at some of the projects being carried out at the Imperial College Internet Centre, formerly the London e-Science Centre, that have been the basis for the development of the work described in this document.

ICENI: ICENI has been in development at Imperial since 2000. 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 [109]. 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. [110] 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 workflows.

GRIDCC: The GRIDCC project was a European FP6 project that finished in September 2007. Imperial College's work was carried out in collaboration with the 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. Work focused on the development of a workflow environment to manage the composition and execution of 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: 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 through various Distributed Resource Managers (DRM). A plugin architecture allows easy development of connectors for deployment on other DRMs.

9.10 Positioning in the Institute activities

The material presented in this paper covers issues related to component specification and composition. It is felt that many of the component technologies within the ICENI architecture are compatible with GCM and that there is direct correlation in many areas. We see ideas and functionality described here as being directly mappable to GCM providing opportunities to extend it with new technologies. The issues related to component composition are of particular significance to workflow layers that sit above GCM components where the security and QoS elements discussed may be linked in with GCM to enhance the results of composing GCM components.

10 Workflows in ProActive/GCM

by N. Ranaldo, E. Zimeo (UoS)

A challenging goal in Grid context is to simplify the application development through the composition and integration of software functionalities, 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 software 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 components. Adopting a simple analogy with the modeling of 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, necessary 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.

Typically component-based systems (such as frameworks based on Fractal component model [70], GridCCM [71] and more recently SCA [72]) 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 behavioral descriptions of the system in space-based compositions, such as in ICENI [73] and in GCM [74].

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” [75]. 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 through the composition of services completely independent from each other and separately implemented adopting different technologies. Such loosely-coupled applications are typically described adopting a 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) [76]. In our opinion, in Grid programming, in which performance and scalability issues are to be 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 functionalities delivered by ProActive objects [77]. It is a preliminary work towards the effective development and efficient deployment of temporarily composed loosely-coupled applications made of GCM components [74]. 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 run-time system, and are the basic building blocks for the development of the GCM-compliant implementation based on ProActive. SAWE [78] was adopted as workflow engine for its features of flexibility and extensibility, and the ProActive Scheduler [79] was integrated in order to achieve dynamic binding of ProActive objects on a pool of resources.

10.1 Contribution

10.1.1 ProActive/GCM and the ProActive Scheduler

GCM (Grid Component Model) [74] 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 [70], a simple and extensible model that enforces the 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, dynamic 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 [80] is proposed in [81]. 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 [82]. The idea is to extend the concept of port for client and server relations to input and output relations in order to model the data control flow, and to use the AGWL workflow language to model the control flow [83]. A reference implementation of the GCM specification is currently under development in the context of the GridCOMP European Project [16], 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 management and synchronization are managed by the run-time system. 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 an automatic and transparent future object mechanism.

Recently a Scheduler, built on top of ProActive, has been developed and released with the version 3.90 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, such as monitoring and fault tolerance. It retrieves available resources by 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.

10.1.2 SAWE

SAWE (Semantic and Autonomic Workflow Engine) is a WES compliant with the WfMC abstract reference model [84], 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 24). 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 assign 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 in the following *ProActive-based activities*).

To describe an application workflow composed of ProActive-based activities, we exploited XPDL extensibility and flexibility features to avoid language modifications 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.

10.1.3 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 types of applications for the interaction with standard technologies, such as Web Services, EJB, form, business rules and standard 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 adopted 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 is responsible to deploy, schedule and monitor the execution of the activity (see Figure 24). 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 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 SAWE Matchmaker

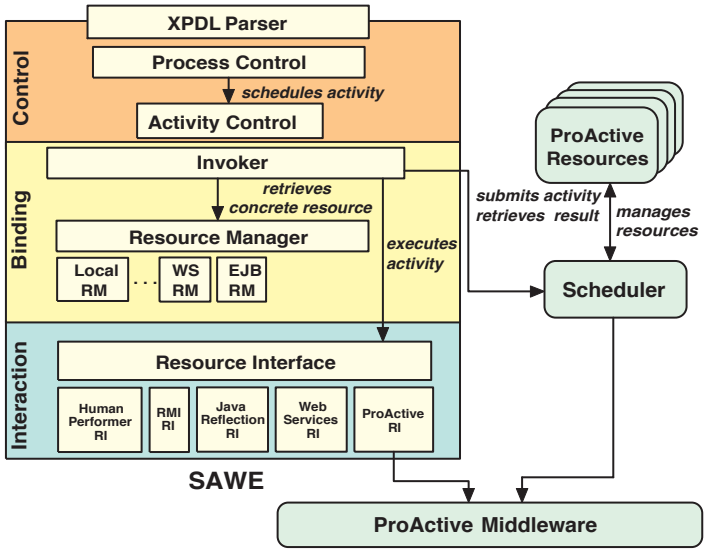

Figure 24: Architecture of SAWE

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.

10.2 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 in order to optimize performance and to satisfy QoS parameters.

11 WP3 activities at UoL

by J. Cunha (UoL)

The UNL (Univ. Nova de Lisboa) group explores multiple dimensions concerning parallel and distributed processing. Here we focus on a description of the more relevant themes to the CoreGRID network, related to software abstractions, tools, and environments for Grid Computing. This research resulted from a collaboration with Professor Omer Rana, Cardiff U., Distributed Collab. Group, since 2002. Jose Cunha and Cecilia Gomes, of Uni. Nova de Lisboa, have been the main researchers involved. We also briefly sketch potential avenues for further research and collaborations involving CoreGRID partners.

11.1 Contribution

Grid environments provide access to diverse heterogeneous resources/services across different organisation boundaries (e.g. high-performance computing; scientific tools; large scientific databases). Due to this, Grid programming is a complex task, requiring:

- Deployment across heterogeneous platforms providing different qualities of service, and with diverse access rights.
- Orchestration of the diverse resources/services to meet the applications' requirements.
- Users with different profiles require diverse levels of transparency to Grid resources' access and control (e.g. transparent access versus deployment and execution control).
- Grid environments are dynamic in nature: unavailability/addition of services/resources at some time; the number of users vary in time; cope with the application requirements' modification on the fly.

Grid Programming still lacks adequate support during the application's life cycle, namely, in the dimensions of: Higher-level abstractions for structured application configuration and coordination; Selective execution control (e.g. suspend and resume part of an application); Reuse of configurations (including behavioural interactions); Dynamic reconfiguration ([111]).

How can we handle the above difficulties? Typically, distributed and Grid applications present similar interaction patterns: a pattern captures common knowledge and experience and describes how a similar set of experiments are to be set-up and managed; it is advantageous to identify those patterns and reuse them.

11.1.1 Design Patterns and Operators for Distributed Grid Environments

In our approach, the goal is to abstract common structures and behaviours, and to integrate the corresponding abstractions into support tools and environments. This work was done in the context of M. Ceclia Gomes PhD Thesis ([112]) under the supervision of Jose Cunha.

The approach consists of:

- developing a model for reuse and flexible composition of typical patterns which capture common usage configurations and interactions:
- based on Structural Patterns, such as Pipeline, Star, Ring, Proxy, Facade, Adapter, etc, and on Behavioural Patterns, such as Master-Slave, Producer-Consumer, Peer-to-peer, etc.
- based on Structural and grouping Operators: Increase, Decrease, Extend, Reduce, Embed, Extract, Group, Rename/Reshape, etc, to increasing and decreasing the number of elements; build hierarchies; replace patterns, etc.
- based on Behavioural Operator: Start, Terminate, Log, Stop, Resume, Restart, Limit, Repeat, etc., allowing to suspend and resume execution; to restart execution periodically; to repeat execution;

Pattern operators are invariant on the semantics of the manipulated patterns (e.g. a star will remain a star after being operated), and independently of the development stage. By considering patterns as first-class entities in several stages of the application life-cycle: development, execution, and reconfiguration, it becomes possible to integrate the model in working prototypes incorporated into software development environments.

Several operator categories provide application manipulation in different dimensions. Also, besides being available through GUI and editors for direct user manipulation at application development time, pattern operators are also amenable to automated invocation through scripts, thus opening a way to possible autonomic control and supervision of application execution and reconfiguration.

The approach can be applied to different steps in the complex cycle of user activities in a problem-solving environment, such as, for example:

1. Problem specification
2. Configuration of the environment including component selection for simulation, control, visualisation
3. Component activation and mapping
4. Initial set up of application parameters
5. Start of execution, possibly with monitoring, visualisation and steering
6. Analysis of intermediate and final results
7. Reconfiguration

A methodology is devised with the following main steps:

1. Deploy Structural Patterns,
2. Refine them through Structural Operators,
3. Use Behavioural Patterns to define control/data flow/interactions,
4. Use Behavioural Operators to manage execution

In such a way, the methodology contributes to building applications in a more structured way: by selecting appropriate set of patterns, combine them according to operator semantics; by defining new patterns and operators found useful and add them to the environment.

As a proof-of-concept, work has been performed to extend the Triana workflow system, in collaboration with Cardiff University, in order to incorporate the mechanisms supporting patterns and operators ([113, 114, 115]).

11.2 Positioning in the Institute activities

From our recent and ongoing research we identify several dimensions that can be considered for joint future research. Overall, this aims at further exploration of the pattern and operators model: integrate into working prototypes and relate to workflow tools, that can be used for application development, and explore their use in support of dynamic reconfiguration; explore relationships and benefits of combining with Grid component and service-oriented models.

Namely the following issues are worth of further research:

- Coordination and dynamic issues related to pattern-driven and operator controlled, run-time reconfiguration as a reaction to requests made on-demand (by the User), or automaticly triggered by the middleware.
- Extensive evaluation of the approach on real applications (concerning expressiveness and scalability)
- Integration with Grid component-models, GCM.

One our current goals is to pursue an identification of possible collaborations with CoreGRID members in the above and related dimensions.

References

[1] Service component architecture, 2007. http://www.ibm.com/developerworks/library/ specification/ws-sca/.

[2] Jboss rules home page, 2008. http://www.jboss.com/products/rules.

[3] Tuscany home page, 2008. http://incubator.apache.org/tuscany/.

[4] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi, and C. Zoccolo. Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST implementation. In *Proceedings of HPC-GECO/Compframe Workshop*, 2006. Paris, HPDC-15 associate workshop.

[5] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on Grids. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, June 2007.

[6] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in gcm: autonomic management of Grid components. In *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, Toulouse, France, February 2008. IEEE.

[7] M. Aldinucci, M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo. High level Grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.

[8] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[10] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[11] M. Cole and A. Benoit. The Edinburgh Skeleton Library home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.

[12] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. Reeve A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe*. Springer Verlag, June 1993. LNCS No. 694.

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

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[15] H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[16] GridComp: Effective Components for the Grids, 2007. `http://gridcomp.ercim.org/`.

[17] ProActive home page, 2007. `http://www-sop.inria.fr/oasis/proactive/`.

[18] J. Serot, D. Ginhac, and J.P. Derutin. SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In *Proceedings of the 5th International Parallel Computing Technologies Conference (PaCT-99)*, September 1999.

[19] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency*, 6(4):68–73, 1998.

[20] W3C. Web services home page, 2006. `http://www.w3.org/2002/ws/`.

[21] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, and C. Zoccolo. Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components. In *Joint Workshop on HPC Grid programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15*, June 2006.

[22] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of Grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, Lisboa, Portugal, August 2005.

[23] Marco Aldinucci, Marco Danelutto, and Marco Vanneschi. Autonomic QoS in ASSIST Grid-Aware Components. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[24] H. Algestam, M. Offesson, and L. Lundberg. Using Components to Increase Maintainability in a Large Telecommunication System. In *Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*.

[25] Françoise Baude, Denis Caromel, Ludovic Henrio, and Paul Naoumenko. A flexible model and implementation of component controllers. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, 2007.

[26] D. Bernholdt, B. Allan, R. Armstrong, F. Bertrand, and K. Chiu. A component architecture for high performance scientific computing. *ACTS Collection special issue, Intl. J. High-Perf. Computing Applications*, 20, 2006.

[27] S. Bouchenak, F. Boyer, E. Cecchet, S. Jean, A. Schmitt, and J.-B. Stefani. A component-based approach to distributed system management - a use case with self-manageable J2EE clusters. In *11th ACM SIGOPS European Workshop, September 2004*.

[28] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience (SP&E), special issue on "Experiences with Auto-adaptive and Reconfigurable Systems*, 36, 2006.

[29] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag New York, Inc., 2005.

[30] CoreGRID Programming Model Virtual Institute. Basic features of the Grid component model (assessed), 2006. Deliverable D.PM.04, CoreGRID, Programming Model Institute.

[31] Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-oriented coordination in mobile ad hoc networks. In Amy L. Murphy and Jan Vitek, editors, *COORDINATION*, volume 4467 of *LNCS*, pages 231–248. Springer, 2007.

[32] P.C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *LNCS*, pages 1–14. Springer-Verlag, 2003.

[33] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 230–254. Springer, 2006.

[34] P. Greenwood and L. Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *Proceedings of Dynamic Aspects Workshop (DAW) (held with AOSD 2004)*. Published as Research Institute for Advanced Computer Science (RIACS) Technical Report 04.01., Lancaster, UK, March 2004.

[35] C. Herault, S. Nemchenko, and S. Lecomte. A Component-Based Transactional Service, Including Advanced Transactional Models. In *Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Revised Selected Papers*, number 3563 in LNCS, 2005.

[36] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[37] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[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.

[39] Paul Marrow and Antonio Manzalini. The CASCADAS Project: a Vision of Autonomic Self-organising Component-ware for ICT Services. In *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS 2006)*.

[40] Vladimir Mencl and Tomas Bures. Microcomponent-Based Component Controllers: A Foundation for Component Aspects. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 729–737. IEEE Computer Society Press.

[41] Arun Mukhija and Martin Glinz. The CASA Approach to Autonomic Applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, pages 173–182.

[42] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, may/june 1999.

[43] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic Grid applications. *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, 9(2), 2006.

[44] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, 2006.

[45] I. Foster, C. Kesselman, *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[46] Massimo Coppola, Marco Danelutto, Sebastien Lacour, Christian Perez, Thierry Priol, Nicola Tonellotto, and Corrado Zoccolo. Towards a common deployment model for Grid systems. In Sergei Gorlatch and Marco Danelutto, editors, *CoreGRID Workshop on Integrated research in Grid Computing*, pages 31–40, Pisa, Italy, November 2005. CoreGRID, IST.

[47] Programming Model Institute. Basic features of the Grid component model (assessed). Technical report, CoreGRID, March 2007. D.PM.04.

[48] Currle-Linde, N., Küster, U., Resch, M., Risio, B.: *Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems*. ParCo 2005, Malaga, Spain, 2005.

[49] N. Currle-Linde, P. Adamidis, M. Resch, F. Bös, J. Pleiss: *GriCoL: A Language for Scientific Grids*, Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing, pp. 62, Amsterdam, December 2006.

[50] M. Baldanzi, M. Coppola, P. Costa, D. Laforenza, G. Pierre, and L. Ricci *Design and Specification of a Prototype Service/Resource Discovery System* XtreemOS Technical Deliverable D3.2.4, Work Package 3.2, November 2007.

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

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

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

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

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

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

[57] Jim Dowling. *The Decentralised Systems Coordination of Self-Adaptive Components for Autonomic Computing Systems*. PhD thesis, University of Dublin, Trinity College, 2004.

[58] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

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

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

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

[62] Fahringer, T., Qin, J., Hainzer, S.: Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and Grid 2005 (CCGrid 2005). Volume 2., Cardiff, UK (May 2005) 676–685

[63] OMG: Unified modeling language. Document formal/2007-02-05 (February 2007)

[64] Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. International Journal of High Performance Computing Applications **20**(2) (2006) 163–202

[65] Yu, J., Buyya, R.: A taxonomy of workflow management systems for Grid computing. Journal of Grid Computing **3**(3-4) (september 2005) 171–200

[66] Taylor, I., Shields, M., Wang, I., Harrison, A.: Visual Grid Workflow in Triana. Journal of Grid Computing **3**(3-4) (September 2005) 153–169

[67] Altintas, I., Birnbaum, A., Baldridge, K.K., Sudholt, W., Miller, M., Amoreira, C., Yohann: A framework for the design and reuse of Grid workflows. In: First Intl. Workshop on Scientific Applications of Grid Computing (SAG'04)), Berlin/Heidelberg, Springer (2005) 120–133

[68] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1. Technical report (May 2003)

[69] OMG: CORBA component model, v4.0. Document formal/2006-04-01 (April 2006)

[70] Bruneton, E., Coupaye, T., Stefani, J.: The Fractal Component Model, version 2.0-3. Technical report, ObjectWeb consortium, (February 2004)

[71] C. Perez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. *International Journal of High Performance Computing Applications (IJHPCA)*, 17(4): 417-429, 2003.

[72] M. Beisiegel, et al. SCA Service Component Architecture - Assembly Model Specification, version 1.0, Technical report, Open Service Oriented Architecture collaboration (OSOA), March 2007.

[73] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *Proc. of UK e-Science All Hands Meeting*, Nottingham, UK, pp. 627-634, Septembet 2003.

[74] CoreGRID, Programming Model Institute. Basic features of the Grid component model (assessed), 2006. Deliverable D.PM.04. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[75] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward Open-World Software: Issue and Challenges. *IEEE Computer*, 39(10):36-43, 2006.

[76] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4):171-200, September 2005.

[77] D. Caromel, C. Delbe, A. Costanzo, and M. Leyton. Proactive: an Integrated Platform for Programming and Running Applications on Grids and p2p Systems. *Computational Methods in Science and Technology*, 12, 2006.

[78] G. Tretola, and E. Zimeo. Activity Pre-Scheduling for Run-Time Optimisation of Grid Workflows. *Journal of Systems Architecture (JSA)*, 2007.

[79] N. Ranaldo, G. Tretola, and E. Zimeo. Integrating SEGL, CALCIUM and SAWE with a ProActive based Scheduler. In *Proceedings of the CoreGRID integration workshop 2008*, Heraklion-Crete, Greece, April 2008.

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

[81] CoreGRID, Programming Model Institute. Innovative Features of the Grid Component Model (with sample case studies): a Technical Survey. *Deliverable D.PM.07*, 2007. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.07.pdf`.

[82] H. Bouziane, C. Perez, and T. Priol. Combining a Software Component Model and a Workflow Language into a Component Model with Spatial and Temporal Compositions. *Technical Report*, INRIA, 2008.

[83] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proc. of the IEEE CCGrid 2005*, Cardiff, UK, pp: 676-685, May 2005.

[84] D. Hollinsworth, The Workflow Reference Model, Workflow Management Coalition, 1994. `http://www.wfmc.org`.

[85] A. Mayer, A.S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington, *Meaning and Behaviour in Grid Oriented Components*, In 3rd International Workshop on Grid Computing, Grid 2002, volume 2536 of Lecture Notes in Computer Science, Baltimore, USA, Nov. 2002.

[86] A. Mayer, S. McGough, N. Furmento, J. Cohen, M. Gulamali, L. Young, A. Afzal, S. Newhouse, and J. Darlington, Component Models and Systems for Grid Applications, in V. Getov and T. Kielmann, editors, volume 1 of CoreGRID series, Chapter ICENI: An Integrated Grid Middleware to Support e-Science, p. 109124. Springer, June 2004.

[87] Workflows for eScience: Scientific Worflows for Grids, A.S. McGough, W. Lee, J. Cohen, E. Katsiri, and J. Darlington, I.J. Taylor, D. Gannon, E. Deelman, and M.S. Shields, editors, Chapter ICENI. June 2007.

[88] A.S.McGough, Asif Akram, Li Guo, Marko Krznaric, Luke Dickens, David Colling, Janusz Martyniak, Roger Powell, Paul Kyberd, Chenxi Huang, Costantinos Kotsokalis, and Panayiotis Tsanakas, GRIDCC: A Real-time Grid workflow system with QoS, Scientific Programming, 15(4):213–234, Dec 2007.

[89] Grid Enabled Remote Instrument with Distributed Control and Computation (GRIDCC) website. http://www.gridcc.org

[90] Open Grid Forum website. http://www.ogf.org

[91] A.Stephen McGough, William Lee, and Shikta Das, A Standards based approach to Enabling Legacy Applications on the Grid, accepted for Future Generation Computer Systems, 2008.

[92] Jorn Altmann, Costas Courcoubetis, John Darlington and Jeremy Cohen, GridEcon The Economic-Enhanced Next-Generation Internet, In Proceedings of Grid Economics and Business Models (GECON2007), EuroPar 2007, LNCS 4685/2007: 188193, 2007, Springer Berlin/Heidelberg.

[93] A.Stephen McGough, Jeremy Cohen, John Darlington, Eleftheria Katsiri, William Lee, Sofia Panagiotidi, and Yash Patel, An End-to-end Workflow Pipeline for Large-scale Grid Computing, Journal of Grid Computing, p. 1–23, Feb. 2006.

[94] J. Darlington, M. Ghanem, Y. Guo, and H.W. To, Guided Resource Organisation in Heterogeneous Parallel Computing, Journal of High Performance Computing, 4(10):13–23, Dec. 1997.

[95] A.S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, Workflow Enactment in ICENI, In UK e-Science All Hands Meeting, p. 894–900, Nottingham, UK, Sep. 2004. ISBN 1-904425-21-6.

[96] N. Furmento, A. Mayer, A.S. McGough, S. Newhouse, T. Field, and J. Darlington, ICENI: Optimisation of Component Applications within a Grid Environment, Journal of Parallel Computing, 28(12):1753–1772, 2002.

[97] Tomasz Guba?a, Marian Bubak, Maciej Malawski, Katarzyna Rycerz: Semantic-Based Grid Workflow Composition, LNCS. 3911/2006, The Second Grid Application and Middleware Workshop (GAMW2005), 651-658.

[98] A.S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young, Making the Grid Predictable through Reservations and Performance Modelling, The Computer Journal, 48(3):358–368, 2005.

[99] A.S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, Performance Architecture within ICENI, In UK e-Science All Hands Meeting, p. 906–911, Nottingham, UK, Sep. 2004. ISBN 1-904425-21-6.

[100] Hong-Linh Truong, Thomas Fahringer, Schahram Dustdar, "Dynamic Instrumentation, Performance Monitoring and Analysis of Grid Scientific Workflows", Journal of Grid Computing, Springer Science+Business Media B.V. Published online since 22 September 2005.

[101] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C.S. Junior and H.-L. Truong, ASKALON: A Tool Set for Cluster and Grid Computing, Concurrency and Computation: Practice and Experience, Vol. 17, Nos. 24, pp. 143169, 2005.

[102] L. Young, A.S. McGough, S. Newhouse, and J. Darlington, Scheduling Architecture and Algorithms within the ICENI Grid Middleware , In UK e-Science All Hands Meeting, p. 5–12, Nottingham, UK, Sep. 2003, ISBN 1-904425-11-9.

[103] A. Afzal, J. Darlington, and A.S. McGough, QoS-Constrained Stochastic Workflow Scheduling in Enterprise and Scientific Grids, In Grid Computing, 7th IEEE/ACM International Conference on, p. 1–8, Barcelona, Spain, Sep. 2006

[104] Ali Afzal, John Darlington, and A.Stephen McGough, Stochastic Workflow Scheduling with QoS Guarantees in Grid Computing Environments, In GCC '06: Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC'06), p. 185–194, Washington, DC, USA, 2006 IEEE Computer Society

[105] Ali Afzal, John Darlington, and A. Stephen McGough, Capacity Planning and Stochastic Scheduling in Large-Scale Grids, In E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, p. 86, Washington, DC, USA, 2006 IEEE Computer Society.

[106] A.S. McGough and A Akram and D Colling and L Guo and C Kotsokalis and M Krznaric and P Kyberd and J Martyniak, Enabling Scientists through Workflow and Quality of Service, In INGRID07: Proceedings of the 2nd International Workshop on Distributed Collaborative Laboratories - Instrumenting the Grid, S. Margherita Ligure Portofino, Italy, Apr. 2007

[107] T. Ferrari and E. Ronchieri and C. Kotsokalis and P. Tsanakas and D. Colling and Y. Hassoun and A.S. McGough and Y. Patel and Chenxi Huang, On Quality of Service Support for Grid Computing, In INGRID07: Proceedings of the 2nd International Workshop on Distributed Collaborative Laboratories - Instrumenting the Grid, S. Margherita Ligure Portofino, Italy, Apr. 2007

[108] A.S.McGough, Asif Akram, Li Guo, Marko Krznaric, Luke Dickens, David Colling, Janusz Martyniak, Roger Powell, Paul Kyberd, Chenxi Huang, Costantinos Kotsokalis, and Panayiotis Tsanakas, GRIDCC: A Real-time Grid workflow system with QoS, Scientific Programming, 15(4):213–234, Dec 2007

[109] Anthony Mayer, Composite Construction of High Performance Scientific Applications,PhD Thesis, Imperial College of Science, Technology and Medicine, University of London, 2001.

[110] M. Bubak, K. Gorka, T. Gubala, M. Malawski, K. Zajac. Component-Based System for Grid Application Workflow Composition, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS 2840/2003, p. 611-618, Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-20149-6.

[111] J. C. Cunha, Omer Rana, (Editors). Grid Computing: Software Environments and Tools. 332 p., ISBN: 978-1-85233-998-2, Springer, 2006.

[112] Cecilia Gomes. Pattern Operators for Grid Environments. PhD Thesis, Universidade Nova de Lisboa, Nov. 2007.

[113] Cecilia Gomes, Omer Rana, Jose Cunha. Extending Grid-based Workflow Tools with Patterns and Operators. Int. Journal of High Performance Computing Apps, 22(2), 2008.

[114] Cecilia Gomes, Omer F. Rana, Jose Cunha. Pattern and Operator based Problem Solving Environments. Euro-Par 2004, LNCS 3149, 960-971, Springer-Verlag, M. Danelutto, D. Laforenza, and M. Vanneschi (Eds), 2004.

[115] M. Cecilia Gomes, Omer F. Rana, Jose Cunha. Pattern Operators for Grid Environments, Scientific Programming Journal 11(2003), 237-261, IOS Press.

[116] CCA Forum. CCA Glossary, 2005.
`http://www.cca-forum.org/glossary`

[117] Cătălin L. Dumitrescu, Dick H.J. Epema, Jan Dünnweber, and Sergei Gorlatch. User-Transparant Scheduling of Structured Parallel Applications in Grid Environments. In *HPC-GECO/CompFrame Workshop - Component and Framework Technology in High-Performance and Scientific Computing, Paris, France*, 2006.

[118] Hashim H. Mohamed and Dick H.J. Epema. The Design and Implementation of the KOALA Co-Allocating Grid Scheduler. In *Proceedings of the European Grid Conference*, pages 640–650, Amsterdam, 2005.

[119] Eduardo Argollo, Michael Claßen, Philipp Claßen, and Martin Griebl. Loop Parallelization for a GRID Master-Worker Framework. In *Proc. CoreGRID Workshop on Grid Programming Model*. CoreGRID Tech. Report TR-0080, June 2007.

[120] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A Grid Service Architecture for Higher-Order Components. In *Proc. IEEE Int. Conf. on Services Computing*, pages 288–294. IEEE Computer Society Press, 2004.

[121] Martin Griebl. Automatic Parallelization of Loop Programs for Distributed Memory architectures, 2004. Habilitation Thesis. Also available as http://www.fmi.uni-passau.de/∼griebl/habil.ps.gz.

[122] Marty Humphrey, Glenn Wasson, Jarek Gawor, Joe Bester, Sam Lang, Ian Foster, Stephen Pickles, Mark Mc Keown, Keith Jackson, Joshua Boverhof, Matt Rodriguez, and Sam Meder. State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, 2005.

[123] Lehrstuhl für Programmierung, Universität Passau. The Polyhedral Loop Parallelizer: LooPo. http://www.fmi.uni-passau.de/loopo/.

[124] Christian Lengauer. Loop Parallelization in the Polytope Model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[125] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Technical report, HP Labs, 2002.

[126] Eduardo Javier Huerta Yero and Marco Aurélio Amaral Henriques. Speedup and Scalability Analysis of Master–Slave Applications on Large Heterogeneous Clusters. *J. Parallel and Distributed Computing*, 67(11):1155–1167, 2007.

[127] A. Bolotov and A. Basukoski. A Clausal Resolution Method for Branching Time Logic ECTL+. Annals of Mathematics and Artificial Intelligence, Springer Verlag, Springer Verlag, 46(3): 235-263 (2006).

[128] A. Basukoski, V. Getov, J. Thiyagalingam, S. Isaiadis. Component-Based Development Environment for Grid Systems: Design and Implementation Making Grids Work, Springer, 2008 (to appear).

[129] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-based Systems Proceedings of the European Workshop on Software Architectures, June, 2005, Springer-Velag LNCS series, Vol 3527, pp 1-18.

[130] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal component Model. Electronic resource: http://fractal.objectweb.org/specification/fractal-specification.pdf. February 2004.

[131] E.A. Strunk and J.C. Knight. Assured Reconfiguration of Embedded Real-Time Software. DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), 2004, p. 367, IEEE Computer Society.

[132] Marco Aldinucci and Marco Danelutto and Peter Kilpatrick. Management in distributed systems: a semi-formal approach. In *Euro-Par 2007, Parallel Processing*, number 4641 in LNCS, pages 651–661. "Springer" Verlag, August 2007.

[133] Jayadev Misra and William R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Software and Systems Modeling*, 2006. DOI 10.1007/s10270-006-0012-1.