Project No. FP6-004265

CoreGRID

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

Network of Excellence

GRID-based Systems for solving complex problems

D.PM.05 – Survey of advanced component programming models

Due date of deliverable: September 30, 2006
Actual submission date: December 19, 2006

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

Organisation name of lead contractor for this deliverable: UNIPI

Revision: *final*

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

Keyword list: Components, programming models, skeletons

Contents

1 Executive summary

This deliverable summarizes the experiences of the Programming model Institute partners participating in Task 3.3 activities. As defined in the original CoreGRID DoW, Task 3.3 ("Advanced programming models") takes the component model result of subtask 3.2 ("Components and Hierarchical Composition") as a starting point and it considers how useful compositions of components can be provided to the grid programmer that are able to handle commonly arising situations in grid application development. Component compositions are studied that:

- can be used to solve common situations
- are parametric in the kind of computation performed, i.e. they can be instantiated to program different applications according to the same implementation schema
- can be optimized to take advantage of additional knowledge concerning the parallel/distributed computation they model in such a way that grid mechanisms can be better exploited and better performance can be achieved.

Task 3.3 also takes into account the study and design of high level, implementation independent programming models, and of (possibly dynamic) program transformation techniques for efficient implementation development. The component technology derived from task 3.2 is used as the building block for these high level programming models.

Task 3.3 is also aimed at investigating the possibility of dynamic adaptation for parallel and distributed components on the grid. Since grid architectures are known to be highly dynamic, using resources efficiently on such architectures is a challenging problem. Software must be able to dynamically react to changes of the underlying execution environment. In order to help developers to create reactive software for the grid, therefore, task 3.3 investigates a model for the automatic adaptation of parallel components.

This document is organized in two parts:

- the first part covers three topics related to component based programming environment development. Two of these topics are related to structured programming environments (ASSIST in Sec. 3.1 and HOCs in Sec. 3.2) that are based on the algorithmic skeleton approach [30]. The third topic (Dynaco in Sec. 3.3) is specifically about adaptation techniques (that are also covered in the ASSIST framework) implemented in a grid component context.
- the second part covers three topics related to the development of formal models and tools for the support of component based grid programming environments. The first approach (the one exploiting the ORC model in 4.1) demonstrates how a formal model, originally developed to model web computations, can be reused in grid component programming. The second (in Sec. 4.2) shows how formal techniques using temporal logic can be used in the same context. Finally, the third (in Sec. 4.3) presents some advances in the Fractal component model [16] and their application to the GCM context.

Overall, the ideas presented in this deliverable all contribute to the design of the GCM (the *Grid Component Model*) that constitutes the main research result of the CoreGRID Programming model Institute. GCM, as defined in the Institute, can be summarized as follows.

Hierarchical

First , GCM is a **hierarchical** component model. This means users of GCM (programmers) have the possibility of programming GCM components as compositions of existing GCM components. The new, composite components programmed in this way are first class components, in that they can be used in every context where non-composite, elementary components can be used, and programmers need not necessarily perceive these components as composite, unless they explicitly want to consider this feature. This property is already present in existing component models. In particular, the Fractal component model assumes components can be hierarchically composed, and this is one of the reasons that led to the Fractal component model being chosen as the reference model upon which GCM would be based.

Advanced comms

GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RCP-like" use/provide (or client/server) ports, GCM allows **data**, **stream** and **event ports** to be used in component interaction. Data ports allow data sharing mechanisms to be implemented. Using data ports, components can express data sharing between components while preserving the ability to properly perform ad hoc optimization of the interaction among components sharing data. Stream ports allow one way data flow communications among components to be implemented. While stream ports can be easily emulated by classical use/provide ports, their explicit inclusion allows much more effective optimizations to be performed in the component run-time support (framework). Event ports may be used to provide asynchronous interaction capabilities to the component framework. Events can be subscribed and generated. Furthermore, events can be used just to synchronize components as well as to synchronize *and* to exchange data while the synchronization takes place.

As regards collective interaction patterns, GCM supports several kinds of collective ports, including those supporting implementation of structured interaction between a single use port and multiple provide ports (multicast collective) and between multiple use ports and a single provide port (gathercast collective). The two parametric (and therefore customizable) interaction mechanisms allow implementation of most (hopefully all) of the interesting collective interaction patterns deriving from the usage of composite (parallel) components. The current definition of GCM does not exclude the possibility of having further collective interaction patterns in the future, should the ones included in the current definition turn out to be insufficient to support commonly used grid component patterns.

Autonomic

GCM is intended to be used in grid contexts, that is in highly dynamic, heterogeneous and networked target architectures. GCM therefore provides several levels of **autonomic managers** in components, that take care of the **non-functional** features of the component programs. GCM components have thus two kind of interfaces: a functional one and a non-functional one. The functional interface includes all those ports contributing to the implementation of the functional features of the component, i.e. those feature directly contributing to the computation of the result expected of the component. The non-functional interface comprises all those ports needed to support the component manager activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in the achievement of the expected (functional) results but not directly involved in actual result computation. Each GCM component therefore contains one or more managers, interacting with other managers in other components via the component's non-functional interfaces and with the managers of the internal components of the same component using the mechanism provided by the GCM component implementation. Each component has a manager whose job it is to ensure efficient execution of the component on the target grid architecture.

ADL

The GCM component program architecture is described using a proper **ADL** (Architecture Description Language) that decouples functional program develop-

ment from the underlying tasks needed to deploy, run and control the components on the component framework. In GCM, the ADL is mostly inherited from the Fractal ADL.

Interoperability

Last but not least, the GCM component model supports **interoperability** at several levels. First, interoperability is guaranteed in terms of the ability to support several grid middle-ware environments as possible platforms on which to implement GCM and, in particular, to host the GCM framework. Second, interoperability is guaranteed by the possibility of wrapping GCM components into standard Web Services, in such a way that the WS framework can benefit from the "services" provided by the GCM framework (passive WS-GCM interoperability). Third, GCM components are allowed to invoke standard Web Service services during their execution (active WS-GCM interoperability). The current definition of GCM does not prevent the extension of the interoperability features to other frameworks in the future.

Compliance levels

GCM supports the features mentioned above according to several compliance levels, in order to allow easy transition to GCM from other existing component frameworks. Lower compliance levels accommodate components that do not support all the features required by the GCM model, but at least can be identified as GCM components with limited support for the GCM features. High compliance levels host full-featured GCM components.

The above mentioned features allow GCM to be viewed in the light of other component models currently available. For example, GCM can be characterized as CCA plus hierarchical composition, advanced communication patterns and autonomic control; or again, it can be ragarded as Fractal plus autonomic control together with advanced communication patterns.

This document is not a standard "research paper": rather, it summarizes the ideas pursued in the context of the Programming model Institute, Task 3.3. Therefore there is not survey of related work, e.g. of other component models or component based programming models developed outside CoreGRID or outside the CoreGRID Programming model Institute. Partners of the Institute contributed to several publications on the subject (see the CoreGRID bibliography web page at the address `http://www.coregrid.net/mambo/component/option,com_wrapper/Itemid,237/`) and a comprehensive survey of component based/related programming models can be found in the Institue roadmaps (D.PM.01 and D.PM.03). Nor do we present a deep motivation of the main Sections in this document. Section 3 presents tools and environments for the implementation of component-based grid programming. Section 4 presents complementary work on abstract models and associated reasoning mechanisms which support the development of the tools and environments of section 3 and indeed, subsequently, grid applications themselves.

2 Introduction

The main goal of the Programming model Institute is to investigate new programming models, exploiting component technology, and suitable for use in implementing grid applications. Therefore, three aspects are investigated within the Institute:

- a component based programming model suitable for managing the details which programmers must typically deal with when programming grid applications;
- programming models and techniques suitable for implementing a single component among those used to build the grid applications;
- programming models and techniques that can be used to provide programmers with advanced programming environments, that is programming environments "raising the level of abstraction" perceived by programmers when implementing their applications.

Each of these topics is covered by one of the three subtasks of the Institute. In particular, all the research activities relating to advanced programming models are performed in the framework of Task 3.3. The goal of this document is to summarize the more relevant activities performed by Institute partners in Task 3.3 "advanced programming models" during the first two years of activity of the Institute.

These activities mostly originated from projects the partners were (and, currently, are) involved in independently of CoreGRID. However, these research activities have been re-targeted or at least re-focussed by partners in the light of the Programming model Institute activities. Therefore, despite the fact that the different sub-sections of these documents clearly summarize research experience mainly performed by a single Institute partner, they have to be considered both preliminary and consequent to the integration activities within the Institute. For example, the ASSIST experience in Section 3.1 or that of HOC in Section 3.2 mainly report on the results achieved at UNIPI and WWU Muenster, respectively, but the results reported here heavily contributed to the development of the GCM as a programming environment supporting advanced programming models. Both ASSIST and HOC exploit structured (skeleton based) programming models while implementing their own component model. In this respect, these two component worlds will coexist within the GCM framework, or, at least, both the environments will be accessible to the Institute and NoE partners to allow experimentation with their "advanced" facilities offered on top of GCM.

This paper is divided into two main parts: the first, Section 3, presents advanced programming models, i.e. programming environments that can be actually (currently) used by programmers and that provide programming abstractions which are higher level than simple (composable) component abstraction as provided by the plain GCM initially defined in D.PM.02 and currently being refined and assessed in D.PM.04. The latter, i.e. Section 4, presents more abstract models that can be used to reason about advanced programming models such as those investigated by the Programming model Institute and by Task 3.3 in particular.

Each of the sub-sections reporting the different experiences relating to advanced programming models and abstract models is organized in two parts: in the first part, a summary of the research activity is presented, with pointers to relevant, more technical and specific references the reader can use to gain a deeper knowledge of the particular topic. In the latter part, the relationships with the research activities of the Programming model Institute, as defined in the Institute roadmaps, are outlined. In this part the reader should appreciate the contribution made by the different research experiences to the overall Programming model research scenario.

As we focus on the partner research activities relating to Task 3.3, this document does not pretend to be exhaustive in its treatment of programming models for grid applications. In particular, some relevant research activities usually taken into account when considering programming models and environments suitable for grid application development are not considered here. The most notable research framework "omitted" is the one dealing with services. Current grid programming environments are strongly promoting the "web/grid service" concept. The activities in the Programming model Institute do not explicitly investigate any kind of service concept. We believe a component can be viewed as a service with some additional information/functionality associated (typically more extensive meta-information related to its life-cycle) and therefore we have concentrated the efforts in the Institute on the component research topics. However, we took into account the fact that more and more possibilities and functionalities are being made available through services. As a consequence, we have defined GCM in such a way that interoperability with the (Web) services world is guaranteed. Although (Web) services and some other relevant concerns are not highlighted in this document, results described here are due in part to partners' activities in these areas. For example, the research activities related to CCA and CCM were considered by the Institute partners. The ASSIST experience reported in Section 3.1 built on the CCA and GrADs results when investigating the possibility of including some kind of autonomic management within each one of the components and in the parallel components in particular; and it exploited the results of CCM when investigating the possibility of providing component interaction mechanisms different from classical use/provide ports. The reader can refer to the more specific cited references to learn more about how these concepts relate to the results discussed here.

	ASSIST	Dynaco/AFPAC	HOC
Main developers	UNIPI	INRIA/IRISA	WWU Meunster
Availability	GPL	LGPL	OGF Incubator Project
Skeleton components	*		*
Autonomic components	*	*	
Interoperabilty	WS, CORBA/CCM		WS
Host languages	C, F77, C++	C, MPI	Java
Heterogenous grid support	*		*

Table 1: Summary of component based advanced programming environments

3 Advanced programming environments

In this Section we include three different research items, all related to the way an advanced programming environment can be made available to the grid application developer. By the term "programming environment" we mean an actual programming environment, that is an environment that can be used to program real applications running on different kinds of grid. This is to distinguish from the "programming models" presented in Section 4. In the latter case, we will present actual abstract models, that is theoretical frameworks that can be used to support implementation of actual programming environments, by allowing modelling and reasoning about their properties.

In sections 3.1 and 3.2 we present two programming environments that are both based on structured programming models and, in particular, on the algorithmic skeleton programming model as originally elaborated by Cole [30, 29], while in Section 3.3 we present a programming framework which can be used to provide programmers with support for (semi-)automatic adaptivity. Each of these three programming environments has been developed by Programming model Institute partners within their own research projects, outside CoreGRID. However, in different ways, the results achieved in these projects have had an impact on the Programming model Institute roadmap (for instance, some of the primitive features of these environments are currently being considered as primitive features of GCM) and have contributed to the Task 3.3 "advanced programming model" research activities within the Institute.

The skeleton programming model considered in both ASSIST and HOC (in Sections 3.1 and 3.2) is particularly important from the Task 3.3 perspective. Skeleton programming models have been around since their introduction in the seminal Phd thesis by Cole [30]. However, skeleton programming models targeting grids have only recently been designed. In a world where grid programmers must explicitly deal with a surfeit of complex, cumbersome and error prone details (program and data placement, resource management, process mapping and scheduling, communication implementation and handling, etc.) down to the "fabric layer", the skeleton approach promises to raise the level of abstraction provided to the grid programmer to that advocated by the invisible grid concept in the NGG documents [49].

3.1 ASSIST

ASSIST represents an *actual* and *advanced* programming environment that allows programmers to easily design and implement Grid-aware applications. It has been mainly developed in the framework of the Grid.it project (see `http://www.grid.it`). In particular, the environment provides applications with a run-time support featuring dynamicity and adaptivity management. As a consequence, the ASSIST environment is able to satisfy user-defined performance levels while, at the same time, relieving application programmers of almost all the burden usually associated with typical grid issues, such as the high dynamicity of resource availability. The Grid.it model represents a step for the ASSIST environment into the world of software components. A Grid.it component is defined in a hierarchical way, exploiting structured parallelism as an efficient composition tool, and includes all the interfaces needed for supporting adaptive computations. Moreover, interactions and wrapping between different component models are also targetted. The definition includes also an infrastructure of entities managing the component execution, and featuring the same hierarchical organisation of applications.

In this section we highlight some important points and features related to the development of the ASSIST programming environment, and its component model that will be useful, in the near future, in the process of designing and implementing

a GCM compliant programming environment.

In the next section we briefly describe the ASSIST programming environment, highlighting aspects related to the adaptive support provided to applications. Then we describe the Grid.it component model. In the final section we list the points of convergence between the Grid.it component model and the GCM specification, describing the experience we gained in the context of our research work, and how ASSIST and Grid.it can be leveraged to obtain a GCM compliant programming environment.

3.1.1 ASSIST Features

ASSIST programs are structured as generic graphs (identified by the **generic** keyword), where nodes can be primitive modules or composite ones and arcs represent typed streams of data. The interactions between modules follows a data-flow semantics. A generic graph can be used as a module in other composite modules. This allows exploitation of streams as a particular composition construct. No constraint is imposed on the form of graphs, though structured graphs, such as those typical of a classical skeleton model, are a notable class of cases that have efficient implementations. Primitive modules can be both sequential or parallel.

A sequential module wraps one or more functions implemented in one of the host languages. A parallel module is expressed by a general-purpose skeleton, called *parmod*. This means that the parmod construct can be tailored, for each application, to specific instances of classical stream parallel and/or data-parallel skeletons, and also to new forms of regular and irregular parallelism. This allows programmers to easily provide solutions to new, uncommon problems and to support specific efficiency needs.

While sequential modules operate according to a straightforward data-flow semantics, a parmod can operate on multiple input streams and multiple output streams, and exploits a richer semantics. Several distribution and collection strategies are provided for the input and output streams, respectively. Moreover, input streams can be controlled in a data-flow or in a non-deterministic manner. The control of non-determinism is important to model several instances of workflow structures, as well as event driven interaction.

The parallel computation expressed in a parmod is decomposed in sequential units assigned to abstract executors called virtual processors (VP). The parmod can have an explicitly defined internal state for the duration of the computation. Parmod state can be decomposed and distributed across virtual processors, according to a chosen naming rule and topology. These features are important in many cases, for example in non-deterministic/reactive computations, as well as in many irregular and dynamic computations.

The exploitation of the structured approach to parallel computing allowed us to define and implement *reconfiguration strategies*, exploiting structural properties of skeletons and parallel programs expressed by the parmod construct. Thus, each parmod has associated a performance model, and a reconfiguration strategy that can be generated automatically whenever a known parallel structure is employed, or that can be programmed directly in all other cases.

The implemented ASSIST support natively provides for a wide range of dynamic adaptation strategies for parallel modules: new processes can be added/removed at run-time within a parmod, and can be migrated across heterogeneous platforms, using a specifically optimised checkpointing strategy. This allows exploitation of remapping strategies of virtual processor sub-computations onto processing nodes (see [3]).

Reconfiguration actions are implemented and optimised for each parmod taking into account its parallel computation semantics. Migration overhead is especially

dependent on the knowledge we can infer from the high-level specification of the computation that a parmod provides.

It is possible to design reconfiguration run-time support exploiting the same synchronisation points already induced by the parallel semantics. Thus, the actual implementation is designed to provide efficient reconfiguration support without introducing further synchronisation points, other than those previously provided for the non-reconfigurable parmod implementation.

The parmod implementation includes also a Module Autonomic Manager (MAM). This receives real-time information on the state of the parmod from the support, and accepts Quality of Service contracts from the user. The task of the application manager is to apply the reconfiguration strategies to satisfy the user requirements. The current implementation provides monitoring information on the performance of the parmod to the application manager, while it can accept only Performance Contracts.

3.1.2 ASSIST components and grid

The ASSIST programming model has been extended with the concept of component in the context of the Grid.it project. Grid.it components feature some notions of hierarchy and autonomicity. These are especially important in the context of Grid platforms, as they provide means to deal with the high levels of dynamicity of resource availability. The definition of components is general enough to allow the wrapping of applications developed in the context of other frameworks (e.g. CCM, WS) in a Grid.it component. The main issue in supporting such wrapping resides in defining different interaction semantics between Grid.it components, e.g. it is possible to wire a component accepting requests in a stream semantics with a client issuing requests in a RMI semantics. The targetted implementation of Grid.it components are ASSIST programs, as their adaptive support is exposed straightforwardly by the component interfaces. The run-time support of the ASSIST environment is implemented on top of a *Grid Abstract Machine* (GAM), that provides as *abstract services* the functionalities needed by the programming environment to support high-performance, component-based Grid-aware applications. These include resource discovery, management and monitoring; component deployment, execution, and wiring; and, routing of communications through networks with private addresses. In particular, wiring among components may be done via buffered ports, enabling multi-site deployment without a strict co-allocation mechanism. Whenever possible, all these services rely on the underlying Grid middleware (e.g. Globus services), which are just abstracted out at the GAM level. In other cases ([1]), GAM services *extend* Grid middleware ones (e.g. providing monitoring).

As mentioned above, the Grid.it model has a recursive, hierarchical definition. The model includes ground components, i.e. components that do not contain other sub-components, and composite components, formed by multiple sub-components. As usual in component models, a component is characterised by *provide* and *use* ports, that allows the functional wiring of components.

The interactions between a composite component and its sub-components, and the control of these, are part of the definition of the model itself. These features are encapsulated in the concept of *non-functional* interfaces. Grid.it non functional interfaces enable introspection and run-time configuration control of the components and exhibit either an RPC behaviour or an event-based one.

The low-level RPC, non-functional interfaces of Grid.it components are:

- request for monitoring measurements;
- describe the current parallel layout of the component;

- apply a user-provided reconfiguration script;
- suspend/resume the component computation;
- stop the computation, releasing all involved grid resources allocated for the component.

Any Grid.it component can directly be asked to report its instantaneous performances or details about its own modules, their location on the grid and current parallel behaviour. They can also be asked to perform a sequence of run-time reconfigurations: e.g. change of parallel degree, processes mapping onto processing elements. Components can explicitly be suspended (and subsequently resumed or stopped), without affecting the semantics of the parallel computation. Gracefully stopping a component is a low-level mechanism to implement stateless migration [3, 2].

The low-level event-based interface, instead, provides for the subscription to continuous performance monitoring measures, at regular time intervals. It allows real-time monitoring of the execution of controlled components.

The user or a software manager may leverage these low-level interfaces to monitor and control the component run-time behaviour. In the latter case, a component together with its manager constitute an *autonomic component*. A Grid.it autonomic component no longer relies on low-level non-functional interfaces, as described above, and replaces them with high-level interfaces for the static or dynamic submission of a QoS contract, and the subscription for QoS contract violations.

Currently, QoS contracts are described by a specific XML file, and include the specification of the processing bandwidth (service time) in stream-based computations, and/or the completion time, which is often more significant for non-stream computations. Such contracts may be subject to constraints on the amount and on the kind of computing resources.

In the most general definition, a Grid.it QoS contract comprises a component QoS goal and the description on how it should be achieved. The concepts that are part of the definition of QoS contracts, or that influence it, are:

- *Performance features:* a set of variables, which can be evaluated from module static information, run-time data, collected through monitoring, and performance model evaluation.
- *Performance model:* a set of relations among *performance features* variables, some of them representing the performance goal.
- *Performance goal:* a set of inequalities involving *performance features*.
- *Deployment annotations* describing resource needs of processes, such as required hardware (platform kind, memory and disk size, network configuration, etc.), required software (O.S., libraries, local services, etc.), and all other strict constraints to enforce correct execution.
- *Adaptation policy:* a reference to the desired adaptation policy chosen among the ones available for the module. Standard adaptation policies are represented as algorithms and embedded within MAM code at compile time.

The Grid.it component model includes also the definition of a hierarchical infrastructures of *managers* that are responsible for dynamically controlling the behaviour of components. The hierarchical organisation of this infrastructure reflects the actual organisation of application components themselves. The managers can be automatically generated by the ASSIST programming environment.

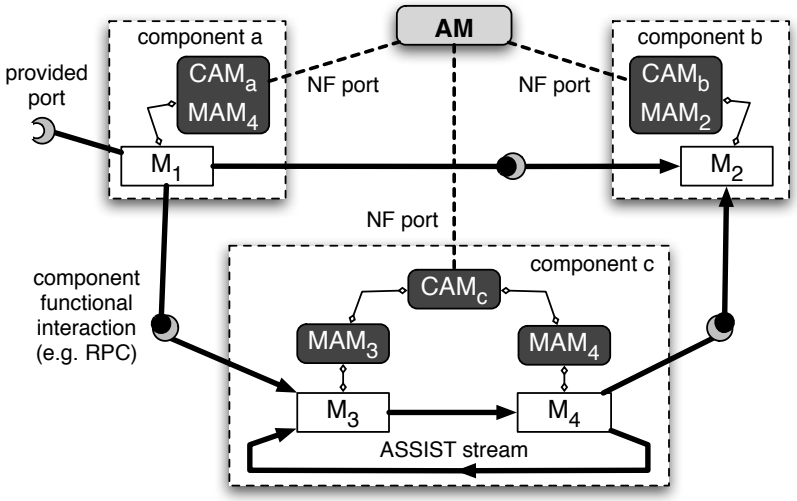

Figure 1: Hierarchical composition of ASSIST managers in a running ASSIST component program

In particular we distinguish: *Module Autonomic Manager* (MAM), *Component Autonomic Manager* (CAM), and *Application Manager* (AM) being the manager of the top component since it (indirectly) controls the whole application (Figure 1 illustrated this manager hierarchy concept).

Each CAM applies control strategies at the level of the associated component, leveraging on non-functional ports of the nested components. The hierarchical CAM interaction obtained in this way is used to implement the abstraction of super-component, which is discussed below.

The CAM can implement strategies based on the dynamic component creation and the wiring functionalities provided by the component model. These allow definition of strategies for managing wrapped legacy components that do not provide native adaptation.

In the most general case, a CAM can receive proposals of restructuring by the child CAMs *(monitor)*. In this case, the CAM has to apply a global performance model in order to detect the need to restructure more children modules and devise a good solution *(analyse & plan)*. Recursively, a CAM can receive reconfiguration requests from parent CAMs, and can send reconfiguration proposals *(execute)*. The root manager (AM) is the eventual agent responsible for the final decisions in the global reconfiguration control.

Overall, an application is described as a *hierarchy* of natural self-governing components that in turn comprise a number of interacting, self-governing components at the next level down. They should continually adapt their configuration to changes in grid resources with the goal of sustaining the requested QoS. Each component should be equipped with a compositional QoS model, in order to orchestrate local reactions to enhance the overall global behaviour.

Among all possible QoS goals, Grid.it managers currently support the performance related ones that are achievable through adaptation within each parallel module. Some of the issues relating to module coordination, as well as application to QoS measures such as reliability, availability, and security, are currently under investigation.

The advantages offered by a hierarchical structure of a component application based on the manager interactions, and the need to coordinate them, suggested the

notion of super-component, i.e. a container that can host both other components or super-components. Super-components are higher-order, parametric components which can be instantiated with other components. They describe common computation paradigms (in fact, skeletons [29]).

The Grid.it support offers the programmer the possibility to describe new or known computation paradigms manually using a graph, possibly including custom code in the related managers. Super-components are well-known parallel structures for which the compiler and support tools can provide suitable performance models and heuristics, and can produce fully-fledged working managers with no programmer intervention.

In the Grid.it model two kinds of super-components are currently defined:

- **DAG**: this enables the wiring of components/super-components as nodes of a Direct Acyclic Graph, as a generalisation of the pipeline parallel pattern, and takes care of automatic load balancing and performance.
- **Farm**: this enables dynamic and adaptive replication of a given host component/super-component, and is functionally equivalent to the replicated component, exposing the same use and providing functional ports as the host. The farm can of course expose different non-functional ports, allowing control of the QoS of the aggregate entity.

These super-components are particularly suited for connection via one-way streams, describing a flow of data that is computed in different logical phases.

Super-components turn the Grid.it component model into a hierarchical component model, in accordance with the GCM specification.

3.1.3 Programming model Institute perspective

The ASSIST programming environment, together with the Grid.it component model, represents a partial implementation of the GCM proposal. The Grid.it implementation constitutes a testbed for several aspects of the GCM specification draft, providing results and opportunities to experiment with key features of the GCM model, such as dynamicity and multi-language/multi-framework interoperability. In the following we underline the contact points with, and main contributions to, the GCM design of the ASSIST and Grid.it research.

Component inner programming model. One of the GCM key issues is that a programming model for the single component has to be provided. The ASSIST programming model allows design of individual parallel components which are adaptive and malleable. The model is user friendly as it is based on a coordination language to espress parallel aspects, with a strong commitment toward modularity and clear definition of interfaces.

Language neutrality. An additional advantage of the coordination-language approach is that ASSIST allows reuse of existing serial code to express the elementary units of computation, and supports multiple serial programming languages at the same time within any application/component. This is a desirable feature, as GCM aims to be a language-neutral model. In designing the GCM standard, the needs of implementations dealing with language interopeability balance those of implementations exploiting a virtual machine based approach.

Interoperability. Along the same lines, GCM targets interoperability with other component frameworks, and ASSIST research has already experimented with support for interoperability among multiple standards, and in particular the CORBA 2, CCM and Web Service ones.

Dynamic reconfigurability. ASSIST's long-time built-in support for dynamic reconfiguration and autonomic management of portable applications is an important source of experimental results for understanding the needs and organization of non-functional interfaces for the single component in the GCM specification. The definition within GCM of several component controller entities, devoted to component autonomic control over Grid execution platforms, will exploit the knowledge gained using Grid.it as a testbed.

Component composition and Hierarchical Management. The same synergy is also evident at the level of component composition. As it raises in GCM the issue of defining and building hierarchies for autonomic control into composite components, it is useful to draw on the experience of Grid.it super-components. Besides turning the Grid.it component model into a hierarchical model, complying with the GCM specification, super-components exploit well-known skeleton patterns and their underlying performance models to achieve autonomic performance control, as well as processing and orchestration of higher-level QoS models and contracts. The very same approach can be applied to GCM component hierarchies, making a substantial contribution in terms of known models, control policies and tested heuristics.

Research on QoS contracts. Already under investigation within ASSIST are means to deal with measures of QoS and SLA related to security, fault tolerance and execution cost, exploiting the same basic mechanisms used to implement autonomic control w.r.t. performance. This work represents a point of contact with the research underlying the GCM definition of controller entities dealing with the various aspects of autonomicity.

Deployment Process and GCM Interfaces. As a programming environment supporting multiple languages, OS's and CPU architectures in the same component, ASSIST can generate quite complex deployment requirements. Several systems of increasing complexity have been developed (ASSISTConf, GEA) to satisfy these contraints within a grid environment, interfacing to Grid.it component managers and supporting anytime component reconfiguration. These experiences, resulting in a deployment process for multi-site, structured parallel applications, can be generalized to the GCM case, providing basic algorithms and a common understanding of component interfaces to interact with the underlying middleware in a portable way.

3.2 Higher-Order Components (HOCs)

Higher-Order Components (HOCs) [48] are software components running on top of a grid middleware which handles the communication amongst them and other software in the grid using standardized portable formats. HOCs got their name from the fact that the input they accept to be sent to them over the network may be data and executable code as well, in analogy with higher-order functions that take other functions as arguments. Specifically aimed at grid programming, HOCs combine two different approaches from software development:

1. **Software Components** [66] are one of the fundamental concepts HOCs originate from: components, such as HOCs, are reusable units of composition, i. e. , not fully self-contained programs, but pieces of software which become useful once they are deployed onto middleware and possibly combined with other components. In most instances, the deployment process consists in packaging the compiled implementation code together with plain text definitions of the components' interfaces (formatted in a middleware-specific style) and copying

these packages into a subdirectory of the middleware installation. Thereby, communication issues (i.e., the encoding and decoding of data into a portable format, for its exchange over the network) are delegated from the components to the middleware which handles them according to the interface definitions. This way, components, which may be implemented in different programming languages and run on different kinds of machines, may interoperate provided that they are hosted using the same kind of middleware.

2. **Algorithmic Skeletons** [30] is the conceptual foundation of the HOC model: a skeleton implements a recurring structure of parallelism in a customizable manner, allowing programmers to take advantage of parallel processing, while they only write the application-specific (sequential) parts of a program. Contrary to the design pattern approach to software development [46] which requires, first, the identification of an applicable program structure for solving a given problem, and, second, the full implementation of this structure, skeletons only require the selection and customization of existing code. Moreover, skeletons are not restricted to a specific programming paradigm (e. g. , object-oriented programming) or target platform type (e. g. , SMP servers).

HOCs make skeletons accessible as components, i. e. , they include the required middleware support, to select an efficient skeleton implementation from a network-accessible repository and to customize it by sending it application code. HOCs simplify grid programming because many grid applications, despite their great variety, exploit recurring algorithmic patterns. Examples of such commonly known, often used patterns are the farm, the pipeline, divide-and-conquer and others. Whereas these patterns may be used in different applications in a slightly different manner, their high-level structure remains mostly unchanged. It is, therefore, advisable to make use of these recurring patterns by pre-packaging their parallel implementations and necessary grid middleware arrangements (consisting in multiple configuration files) together as HOCs in the HOC repository.

3.2.1 HOCs and the Grid

HOCs bridge the gap between grid middleware and grid applications. This challenge is accomplished by the HOC-Service Architecture (HOC-SA) [37], the runtime environment for HOCs, which abstracts over the middleware used in the grid. The most recent HOC-SA implementation uses the Web service resource framework (WSRF) [56] as its middleware and, thus, Web services for interconnecting HOCs and applications. HOCs and the applications using them are distributed across the Internet, thus, forming a grid, where hosts of heterogeneous architectures can be combined for distributed processing. The HOC-SA serves for a separation of concerns: All the technical concerns, related to the required WSRF configuration for making the HOCs accessible via Web services are readily provided by the HOC-SA, while HOC users only deal with application-specific concerns. HOCs allow their users to benefit from the grid middleware to the extent that they can run tasks on remote servers without having to care about how data is exchanged with these servers. Not even the number of executing servers must be known to the HOC users.

The process of grid application programming using HOCs is shown in Fig. 2. The programming and middleware arrangement tasks are divided between two groups of programmers: grid experts and application programmers. While the former (right) prepare the necessary implementations and arrangements for HOCs, the latter (left) develop applications using these HOCs.

For the application programmer, program development proceeds as follows:

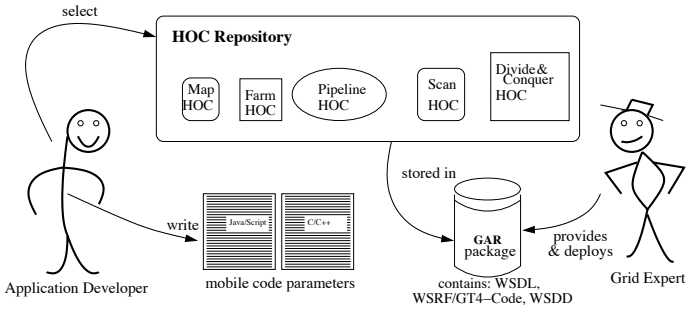

Figure 2: Using HOCs: The Idea

- Select suitable HOCs for the application from the HOC repository;
- Express the application by composing HOCs and customizing them with application-specific code parameters expressed, e.g., as Java code or in a script language such as Ruby that is interpreted on the remote server (the treatment of native code parameters in HOCs is discussed in [36]);
- Rely on the pre-packaged implementations and middleware configuration code of the selected HOCs, packaged as Grid Application Archive (GAR) in the HOC repository.

The person "with a hat" in the figure, is the HOC developer, who is an expert in grid middleware set-up. It is the grid expert who in the HOC approach frees the application programmer from much of the burden related to the necessary middleware arrangements. The grid expert develops the implementation of each HOC, including the necessary Web service configuration for accessing them remotely in the grid. Note that the task of the grid expert in arranging middleware in Fig. 2 is even simpler than it is for an application programmer without HOCs: whereas the expert prepares the arrangement once for each HOC, the programmer had to do it again and again for each application and even for each new version of the runtime environment or the same application.

The code mobility mechanism

In the grid context, a HOC and its code parameters usually reside on different machines of a grid, so code parameters must be *mobile*, i.e. transferable over the network.

Handling the transfer of code using current grid middleware is a technical challenge, since Web services technology, as used, e.g., in the WSRF middleware, provides only limited support for code transmissions: a job submission system, like e. g. , the Globus Resource Allocation Manager (GRAM) [56] or the Sun Grid Engine [65], allows remote execution of any code that represents full programs, i. e. , code that can be executed without a specific interface by calling it directly from the Unix shell of the target server. When a Web service should be used to transmit such code, the code is declared as raw data and encoded for the transmission in the same manner. By contrast, the customizing code parameters for HOCs are specific component codes, i. e. , they are not programs but rather collections of subroutines (or methods in OOP) which only serve a purpose within a particular context like, e. g. , evaluating a pipeline stage or the division predicate of a divide-and-conquer algorithm. These subroutines have different signatures and locations within a code parameter, and, thus, executing them requires for each different parameter a different interface giving the single subroutine access points, i. e. , their addresses/names

and their signatures. However, defining types via interfaces of their own is not supported within a Web service interface definition, where the code parameters must be defined for allowing their exchange over the network.

The HOC-SA provides the HOC developer with two specific technologies providing a workaround for the described parameter range limitation of Web services: the *code service* and the *remote code loader* which enable code mobility in the grid. While the code service is directly used by the clients to upload customizing code for a HOC, the remote code loader is only used by the HOC developer and hidden from the HOC user.

The code service

The code service is a Web service connected to a database for storing and sharing code among distributed computers. It offers two operations: 1) a client can *upload* a code parameter which is then stored in the database and assigned to one unique identifier which is returned to the client. Besides the code parameter itself, the upload operation takes as its input another identifier, which refers to the required interface for running the uploaded code. This identifier is not necessarily unique for each parameter, but unique for one interface, as many code parameters can be accessed using the same type of interface. There is only a limited set of such interfaces (which corresponds to the number of HOCs in the HOC repository) and, therefore, the interface definitions themselves are not sent over the network, but they reside at the server side and clients refer to them using the type identifiers. 2) a server can retrieve a code parameter from the code service using its *download* operation which includes an interface detection mechanism. When the download operation is passed a code parameter identifier, it copies the respective code parameter to a buffer accessible by the server. Moreover, the download operation returns the type identifier of the interface required for running the downloaded code parameter to the calling server.

The remote code loader

The remote code loader is used locally on the servers hosting the HOCs and makes the uploading and downloading of code via the code service transparent to the HOC developer. It downloads code by invoking the download operation of the code service, which supplies the code in a buffer, as if it was some primitive data. The interface identifier returned by the download operation determines the required interface for running the downloaded code. Using this interface (which is locally available to the server) the remote code loader converts the data back into executable code by applying a type cast. In a Java-based HOC, the standard Java class loader is replaced by the remote code loader, allowing the HOC to use classes uploaded by a client as if they were local classes. In the case of C/C++, the interface is defined by a collection of function prototypes and the *gateway* mechanism described in [36] enables location transparency of the actual implementation for MPI-based code in a manner similar to how it is handled in Java.

3.2.2 Programming model Institute perspectives

There is only little divergence between the GCM specification and the HOC model.

Adaptations of Components

The most notable point is a different notion of the term *component adaptation*:
1) In the GCM specification, component adaptation means *platform adaptation*, i. e. , a component is adapted to its execution platform at runtime (e. g. , by migrating code, once a server slows down due to heavy processor load or bandwidth variations). 2) In the context of HOCs, the term adaptation has been used to describe *application adaptations*, i. e. , a component is adapted to a particular ap-

plication [39]: whenever a component is needed that is not readily provided in the HOC repository, an adaptation code can be sent as a code parameter to a HOC which almost offers the required functionality and alter this HOC's behavior according to the application requirements (e. g. , by re-ordering tasks or by running additional processes/threads).

Both, platform adaptation and application adaptation are required in the grid context: platform adaptation addresses the problem that the grid is a highly dynamic platform. Application adaptation addresses the problem, that there is such a great variety of possible grid applications that the HOC repository can never hold adequate HOCs for every conceivable grid application.

Fortunately, this conflict is only a clash of terms, while both technologies are complementary. Using the KOALA scheduler, it has been shown [35] that monitoring information from the Globus MDS [56] can be used to perform platform adaptations of HOCs in a similar manner than it is described in the GCM specification.

An ongoing CoreGRID fellowship between the University of Münster and the University of Passau ("Middleware for smart code distribution" granted to Eduardo Argollo de Oliveira Dias) currently deals with an attempt to use loop parallelization techniques [52] for making application adaptations transparent to the user. The goal of this project is making components so flexible that no specific HOC is needed anymore for every particular application. Instead, advanced HOCs are developed which adapt themselves automatically when needed and which free the programmer from dealing with application adaptation. Anyway, this is future work, and in the current documents, occurrences of the term adaptation should be explained to avoid conflicts.

Enabling HOC Features for the GCM

The main advantages of every HOC are the combination of the interoperability features enabled by the middleware, as explained in Section 3.2.1, with the flexible skeleton approach, described in the front of Section 3.2. Since this kind of synergy is also desirable for the GCM, the current GCM specification (CoreGRID D.PM.02) includes Section 4.4 on "Higher-Order Components and Code Mobility".

The GCM specification requires that the GCM programmer may use Higher-Order Components, which can be accessed via Web services and take data and also code as their parameters, but it is not specified how this support should be technically realized. The GCM specification is mainly adopted from the Fractal model [16] which focuses on composing components together and nesting them using component controllers. The most recent Fractal implementation (the current GCM prototype) uses ProActive [31] as its middleware. The connection to other grid middleware is given by a dynamic component exposure mechanism which allows automatic deployment of Fractal components onto, e. g. , the WSRF middleware [56], where they can be accessed via Web services.

Unfortunately, this convenient mechanism is limited to components which take only primitive parameters. In the CoreGRID workshop on grid systems, tools and environments, it has been shown that the remote code loader and the code service are tools which are independent from the component implementation [38]. They can enable the use of Higher-Order Components in the context of Fractal/GCM as well. In this deliverable, we therefore propose to include the remote code loader and the code service from the HOC-SA in the GCM. We note that there is another ongoing effort to include these tools into the popular WSRF implementation available from the Globus Alliance within the scope of the Globus incubator process [4].

In the remainder of this section, we summarize the main contributions from [38], explaining the role of the code service and the remote code loader in the context of

the GCM.

Integrating the Code Service and the Remote Code Loader in the GCM

The Fractal implementation, which serves as the current GCM prototype, uses the Axis [6] library to generate WSDL descriptions and the Apache SOAP [5] engine to deploy Web services automatically. Service invocations are routed through a custom *provider* (a part of the ProActive framework). When a GCM component should be exposed as a Web service, the user simply calls the static library method `exposeComponentAsWebService`, which generates the required service configuration and makes a new Web service available. The URL of this new service is specified as a method parameter.

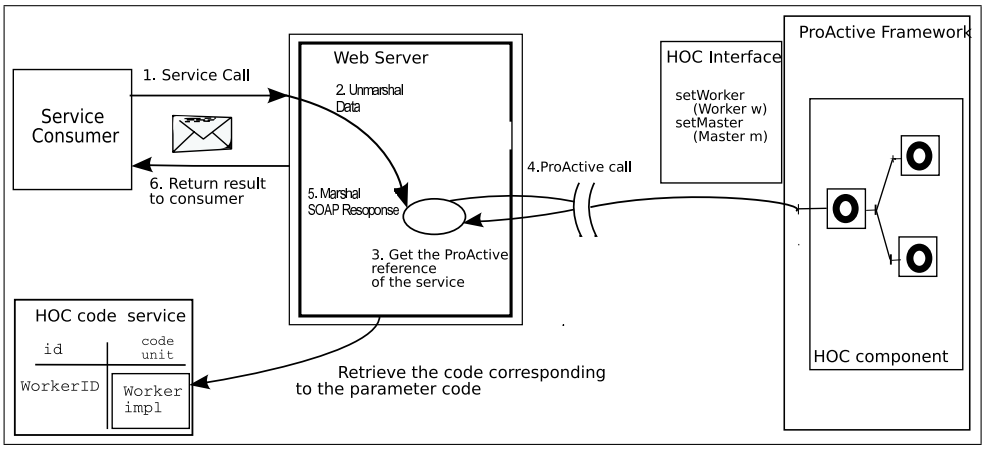

Figure 3: Fractal's Web services mechanism with HOC remote code loading

The Web services automatically created from GCM components do not require the processing of SOAP messages in the code written by the GCM users: when a Web service call (Fig. 3, step 1) reaches the provider, the Apache SOAP engine has already unmarshalled the message and knows which method to call on which object (step 2). Only the logic required to serve the requested operation must be implemented when a new service should be provided. Specifically, the provider gets a remote reference to the targeted interface (step 3), it performs a call from the Web server side to the remote side using the reference (step 4), and it returns the result to the SOAP engine. The engine then marshals a new SOAP message and sends it back to the service consumer (steps 5 and 6).

Whenever the `exposeComponentAsWebService` method is applied to a HOC, parameters of complex types must be transferred indirectly by uploading them to the code service and passing only references to them (e. g. , the `WorkerID` in Fig. 3) to the newly created Web service, as explained in Section 3.2.1. A practical way to develop a HOC that complies with the GCM specification is deriving a `HOC`-class from the base class for defining components in the GCM (the Fractal `Component`-class). Inside the ProActive provider, Java's `instanceof` operator is used to detect this `HOC` type and all non-primitive parameters of the HOC are then simply replaced by the `xsd:string` type for declaring the parameter identifiers in the Web service interface definition. In the server-sided HOC code, the remote code loader is used to obtain an `Object`-type instance of the class corresponding to such an identifier. Any of these `Objects` is cast there appropriately, for making it available, e. g. , as the `Worker` in the Farm-HOC.

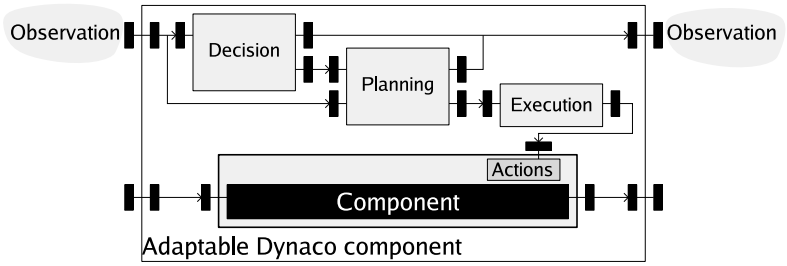

Figure 4: Instantiating the Dynaco framework within the controller of a component makes that component adaptable.

In summary, our extensions of the GCM Web service creation mechanism involves two steps:

- First, we generate a WSDL description that maps behavioral parameters (the code carrying ones) to identifiers for denoting the respective code in the HOC-SA code service.
- Second, we extend the routing in the service provider to retrieve the right code unit according to the identifier sent by the client (Fig. 3, step 2.1) and use the HOC-SA remote code loader for instantiating the parameter via reflection, i. e. , inside the GCM implementation there is no need to differentiate between primitive data and behavioral parameters.

3.2.3 Future Work

An experimental integration of HOCs and Fractal has already been accomplished [38]. However, there is more potential for integration activities for HOCs and the GCM.

In the currently available HOC implementations [61], the HOC repository is dispersed over the grid and there is no lookup directory. The HOC developer has to know which HOCs are hosted by which server and also the Web service URLs for accessing them. Recent research on software ontology, match-making for Web services and also the ASSIST approach towards the "invisible grid" described in Section 3.1, may help to improve this situation in the future.

3.3 Dynaco

Dynaco is a design method and a framework developed at INRIA. It is intended to provide tools for implementing adaptability in the context of component-based software engineering. From the design phase to the execution, Dynaco helps developers in making adaptable their own components, including those encapsulating legacy codes.

In the context of Dynaco, adaptability is an ability that is given to individual components to adapt themselves to their context. Components are thus able to modify themselves and the way they work in order to better fit their execution context. Furthermore, when the execution context evolves, components evolve consequently, at runtime, in such a way they continuously remain best fitted.

As depicted on Figure 4, Dynaco defines a structural model of adaptability. It defines a collection of subcomponents, which provides adaptability when assembled together. Defining clearly the responsabilities and interfaces of those subcomponents helps developers in designing adaptability. Basically, adaptability is decomposed into four major functionalities:

- The *observation* of the context (including the execution platform and the component itself) permits the component to track changes of the parameters to which it adapts.
- Given the observations, an adaptable component has to *decide* whether the changes that are observed are significant enough to make an adaptation worthwhile. If it is the case, the component needs to determine which configuration should be used accordingly to the observations.
- In order to achieve decisions that have been taken, the component has to *plan* which actions must be performed. In this phase, low-level mechanisms are assembled in a runtime generated program, which transforms the component to make it adopt the particular configuration that has been decided. The low-level mechanisms may be implemented into the component; they can also be added when the component is made adaptable.
- In the end, the component should *execute* the actions that have been identified. More precisely, the actions should be scheduled in such a way that it takes into account the progress of ongoing executions within the component.

This decomposition is similar to a materialization of the control loop envisioned by Kephart and Chess for autonomic computing [50], while specifying more precisely roles and responsabilities. It focuses on separating as much as possible the concerns depending on their dependencies to the environment, to the adaptation or to the component.

From the implementation point of view, Dynaco is a component framework. Developers are expected to specialize it to their actual components and adaptations. A *policy* indicates how decisions are made for each individual component. A *guide* maps decisions to plans. And several *actions* are implemented in order to be used by plans. The Dynaco framework does not specify the formalisms in which those subcomponents should be expressed. It rather defines placeholders for reusable generic engines for decision-making, planning and execution. Developers are thus given the ability to choose the most suitable formalism depending on their actual needs. For instance, the decision-making engine can be a simple delegating component to a policy expressed thanks to classical programming languages; an expert system engine that uses a rule-based policy; a generic optimization engine (such as a genetic algorithm) that expects an objective function as a policy; a machine learning algorithm; and so on. The framework approach and the ability to highly customize it make Dynaco generic enough to be virtually suitable to any adaptation and any component.

In addition, Dynaco provides an implementation of its *execution* subcomponent for parallel components: Afpac [25]. Based upon a general execution model of adaptation in the context of parallel executions, Afpac applies to the particular case of SPMD codes. It defines a criterion for the global states from which the adaptation occurs; and it includes an algorithm for finding such global states amongst the upcoming ones. Basically, the idea of Afpac's criterion consists in considering the adaptation as a collective operation. Consequently, adaptation actions are implemented with the same SPMD paradigm as the component itself. The criterion thus matches the ideas of the SPMD paradigm.

Integrating adaptability within a component requires that the execution flow can be intercepted, in order to suspend the component and execute the adaptation. Furthermore, Afpac requires to track the progress of the execution along the control structures. Afpac suggests a semi-automated approach to the integration of adaptability: developers are expected to manually insert calls to Afpac when the execution flow can be intercepted; then automatic program transformation inserts

additional calls at enclosing control structures. Thanks to this approach, Dynaco and Afpac can be successfully used to make adaptable legacy components, without requiring any restructuring of the code. Nevertheless, it assumes a manipulable representation of the components' programs, such as their source code.

In order to make things clearer, here follows an example scenario of using Dynaco. Given an existing parallel component, it may be desirable to make it adaptable to the actual number of processors: the component should execute as much processes as processors. Developers then pick a monitor that tracks the number of available processors in the system. They also compute the performance model of the component as a function parameterized by the number of processes. Developers reuse an existing genetic algorithm component as the decision-making engine; the policy is the performance model of the component. Two reactions can be envisaged: spawning new processes or terminating some of the running ones. Developers can thus choose a simple match-and-map function as the planning engine. In addition, developers are expected to implement the two subprograms that respectively spawn and terminate processes. To terminate, developers can choose Afpac as the plan execution engine. Consequently, they have to insert a statement at each code position where the two subprograms for spawning and terminating processes can be executed. Those statements are added to the component's program. Additional required statements are added thanks to the source-to-source transformation tool provided by Dynaco. Finally, inserting the Dynaco framework instance within the controller makes the component effectively adaptable, as shown in Figure 4.

3.3.1 Dynaco components and grid

The several existing grids show that resource availability is not constant over time within such platforms. Of course, the high amount of resources reduces drastically the mean time between each failure. Nevertheless, faults are not the only reason to varying resource availability. Grids are platforms that are shared amongst several users: there is usually no exclusive access to such a platform. Users are thus allowed to submit and execute their own applications concurrently. As concurrent applications consume some of the resources, activities of the users impairs resource availability. In addition, grids are commonly built by pooling resources from distinct institutions. Moreover, each single institution may participate to several grids. Consequently, there is usually no unique political and administrative head to the whole platform. It results in more frequent administrative tasks such as software updates and hardware upgrades.

Experiments have shown that applications in the context of grid computing can perform better if they take into account those variations, i.e. if they adapt dynamically. Indeed, if they do so, they may execute faster when more resources are available, while avoiding the requirement for fault tolerance techniques when resources disappear.

Fault tolerance techniques permit components to deal with sudden resource disappearance. However, they may be costly in other circumstances. For instance, preventive checkpointing should not be required to take into account resource appearance. The approach proposed by Dynaco aims at providing a lightweight mechanism for handling resource availability variations. Indeed, Dynaco suggests that adaptation is performed without stopping, checkpointing nor restarting the component. To do so, Dynaco focuses on cases when the component can continue its execution after being notified of the variation: for instance, when an administrator updates some software, components can be notified in advance, even before the affected resources become effectively unavailable.

In addition, the Afpac implementation of the *execution* subcomponent of Dynaco allows the adaptation of parallel components. Thanks to this, dynamic adaptability

is made available to scientific applications. Dynaco is thus applicable in the context of grids.

Dynaco has been designed with component-based software engineering in mind. Furthermore, the framework has been implemented for the Fractal component model. First, the framework itself is an assembly of components: each function described previously is one component. Second, the framework is designed to be integrated in part within the controller of adaptable components. In particular, the execution of plans should be a part of the controller in order to get sufficient access to the content of the components to be able to execute the adaptation actions. On the other hand, outsourcing the observations of the environment permits to factorize this function when several adaptable components need to collect the same information. The status of the other functions (decision-making and planning) is not constrained by Dynaco.

3.3.2 Programming Model Institute perspective

Adaptability has already been mentioned in several preceding deliverables of CoreGRID and of the Programming Model Institute, sometimes with a different terminology (adaptivity, autonomic component). In particular, D.PM.02 (Proposals for a Grid Component Model) cites adaptability as one of the expected GCM general features. The institute has agreed in this deliverable that components should be able to change their configuration at runtime in order to react to dynamic resource behaviour; to reconfigure themselves upon fault recovery; and to evolve depending on data's, parameters', and other components' characteristics. It also refers to the functionalities envisaged by Kephart and Chess for autonomic computing [50] (self-configuring, self-healing, self-optimising and self-protecting).

Within the institute, activities about adaptability appear in tasks 3.2 and 3.3 according to D.PM.03 (Roadmap version 2 on programming model). In task 3.2, the institute agreed to provide specifications for controllers for the adaptability and autonomy of components. Several levels of autonomy are expected to be specified. In task 3.3, the institute proposes to work about autonomic control of adaptable components, and in particular about the interactions with the underlying middleware.

At last, in D.STM.02 (JPA 2), the Programming Model Institute identified adaptability as the topic of one of the fine-grain activities. This activity concerns mainly task 3.3, but also task 3.2.

This short review of the institute deliverables shows well that the institute has considered adaptability as one of its topics since the early stages. Several scenarios of GCM component adaptation have been envisioned as motivations. Dynaco may provide the institute with some initial experience to achieve its goals. Furthermore, Afpac, as a part of Dynaco, gives some reusable basis for the adaptability of GCM components, which are expected to encapsulate parallel codes as stated in the previous deliverables.

In addition, the design and implementation of Dynaco rely on the Fractal component model, which also serves as a reference for GCM. Furthermore, both Dynaco and GCM consider that components can encapsulate parallel codes, such as those using MPI.

Unlike GCM as described in D.PM.02, Dynaco makes no distinction between different types of adaptability (i.e. the self-∗ functionalities mentioned in D.PM.02 and previously recalled). Furthermore, while specifications in D.PM.02 list a collection of functionalities (self-configuring, self-healing, self-optimising and self-protecting), Dynaco focuses on specifying how component developers can implement and integrate adaptability within their components. The institute should consider whether Dynaco's approach is relevant for GCM specifications or not.

Experience resulting from the work on Dynaco shows that most of the usual software engineering techniques do not provide fully satisfactory solution to the problem of integrating adaptability. In the case of the Afpac part of Dynaco, it is suggested that integration should be done by semi-automated program transformation. This approach assumes a manipulable representation of the components' programs, such as their source code. As GCM is intended to be language independent, the assumptions made by Afpac do not hold in the case of GCM. Consequently, the proposed approach is not applicable to GCM. The institute should thus investigate different solutions to integrate adaptability.

Finally, Dynaco shows that component-based architectures are a valuable tool for structuring the adaptabilty itself. The institute should propose a solution to allow developers to work on the adaptability architecture separately from the applicative one.

4 Abstract support models and tools

In this Section we present the results achieved by Programming model Institute partners in the framework of Task 3.3 activities, in particular those results related to abstract, theoretical models suitable for modelling advanced grid programming environments and techniques.

In particular, Section 4.1 is related to the research activity performed by QUB and UPC partners on the ORC model by Misra and Cook, Section 4.2 concerns the work performed at UoW investigating the possiblities offered by different kinds of temporal logic systems, and finally Section 4.3 is about the work performed at INRIA on specific abstract models used to represent behavioral semantics of realistic distributed programming frameworks, including component based ones.

Overall, the three contributions selected here should give the reader a precise idea of the way theoretical results can be used to improve research in the field of "advanced programming models" for grids based on component models such as GCM, and give the reader an overview of Institute activities in this area.

It is worth pointing out, in this case, how the research items discussed in Section 4.1 already represent a joint research effort by two partners of the Programming model Insitute (QUB and UPC) and a currently emerging joint activity with a third partner (UNIPI) that also exploits a six months REP grant, and how the research activity on temporal logic performed at UoW is currently (since the London meeting in January 2006) being considered for common investigation with the QUB and UPC groups. This significantly contributes to the integration of research efforts of the Institute partners. Moreover, part of the research results described in Section 4.3 are currently being considered for direct inclusion in the "assessed GCM" deliverable D.PM.05.

4.1 ORC

Collaborative work between QUB and UPC has investigated the modelling of grid systems with the aim of creating models which capture the essential grid-specific properties of applications, i.e. properties which relate to, for example, resource discovery, dynamic adaptivity in response to connection or site failure, load balancing, etc., rather than properties of the application domains *per se*. A preliminary abstract model of a grid has been developed [64] and used in conjunction with the ORC language to represent some typical grid operations.

ORC [55] was introduced by J. Misra and R. Cook in 2004 as a language for specifying the orchestration of distributed services. A brief summary of ORC is given here. It is based on the idea of a site call. In ORC all operations must be realised as site calls (e.g. there are no in-built arithmetic operations - addition of x and y may be simulated by the site call $add(x, y)$). In general a site call M may update the recipient site which, in turn, may call other sites and reply. A fundamental concept of ORC is that a site call may fail (i.e. the sender may not receive a reply). This may be due to a faulty network (either the outgoing or incoming message may fail) or to the recipient site being down. There are some special sites:

- 0 never responds (0 can be used to terminate execution of threads);
- if b returns a signal if b is true and remains silent otherwise;
- $Rtimer(t)$, always responds after t time units;
- *let* always returns (publishes) its argument.

In addition, calls made to the generic grid site are also are considered.

ORC site calls may be orchestrated using expressions. The simplest expression is a site call, possibly with parameters. More complex expressions can be constructed as follows, where E_1 and E_2 are expressions:

1. operator $>$ (sequential composition)
   $E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x. If E_1 produces two results, say x and y, then E_2 is evaluated twice, once with argument x and once with argument y. The abreviation $E_1 \gg E_2$ is used for $E_1 > x > E_2$ when evaluation of E_2 is independent of x.

2. operator $|$ (parallel composition)
   $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. Both evaluations may produce replies. Evaluation of the expression returns the merged output streams of E_1 and E_2.

3. where (asymmetric parallel composition)
   E_1 where $x :\in E_2$ begins evaluation of both E_1 and $x :\in E_2$ in parallel. Expression E_1 may name x in some of its site calls. Evaluation of E_1 may proceed until a dependency on x is encountered; evaluation is then delayed. The first value delivered by E_2 is returned in x; evaluation of E_1 can proceed and the thread E_2 is halted.

4.1.1 Components and grids with ORC

In this section it is first shown how a grid may be modelled in ORC; then an abstract component model is presented and it is shown how placement of components on a grid may be specified.

Grids with ORC A grid is modelled as a collection of interconnected sites. A user may wish to submit a software component(s) to a grid for execution. To do this a means of navigating the grid (in order to find a suitable site for component placement) is required. In this section a generic definition of grid site is proposed.

A site is defined to be a sextuple comprising a unique name (N), a set of components, or jobs, awaiting execution (C), a collection of services (S) that can be utilised by users, a directory (ID) providing information about grid sites, an engine (E) which has the potential to execute components to produce results, and a local manager (M) which co-ordinates workflow.

$$gridsite = (N\ ,\ C\ ,\ S\ ,\ ID\ ,\ E\ ,\ M)$$

In a particular site only some of these fields may be instantiated. For example, a "yellow pages" information site may contain only a site name, an information directory and a manager:

$$yellow\_pages = (yp,\ ,\ ,id,\ ,m1)$$

Here id provides information about other sites (for example, the set of sites which offer the capability to execute FORTRAN 90 programs). This information may become obsolete or unreliable after a period of time. The manager $m1$ may interact with other sites and update id as appropriate.

A supercomputer centre offering services, S, (compilers, operating systems, standard libraries such as SCALAPACK etc.) will typically contain a set of jobs awaiting

execution, J, a supercomputer, $super1$, for generating results, and a manager. This may be expressed as:

$$super\_computer\_centre = (scc, J, S, \ , super1, m2)$$

Here the manager $m2$ interacts with users, accepts jobs for execution, manages the"queue" of jobs and returns results to users.

Users wishing to submit software artifacts for execution on the grid can themselves be regarded as sites with components and managers.

$$user = (u, C, \ , \ , \ , m3)$$

Here the manager $m3$ is software for placing a set of components C on the grid for execution. An example of such a manager is developed later.

Site Failure. In order to see how ORC and grid components are used together let us consider a site failure. In the case of site failure (observed as silence) a user may remain waiting for a response (non-termination). Silence may be temporary – due to a delay in response from the target site – or permanent. Evaluation of some ORC expressions *must* succeed. For example, given a grid site g the expression

$$Terminate = \ \mathsf{let}(g) \ \ \mathsf{where}\ g :\in \{N \mid Rtimer(100) \gg let(stop)\}$$

has one thread $Rtimer(100) \gg let(stop)$ comprising local site calls only. These calls must succeed and so evaluation of the expression must terminate. However, this is not the case for an arbitrary site call. Grid programs should react when a response, which they are awaiting, is not forthcoming. Typically, if there is no response to a site call within a specified period then an exception handling routine will be invoked. One way of dealing with site calls that do not respond is to repoll the site. Thus, instead of conducting a single poll at once, polls may be carried out at regularly or irregularly spaced intervals. Two such polls, $Poll^*$ and $TPoll$, which repeatedly instantiate an ORC expression, E, are defined below:

$$Poll^*(E) = \ \mathsf{let(r)} \ \ \mathsf{where}\ r :\in \{ \mid_{t \in \mathcal{N}} Rtimer(t) \gg E\}$$

Here E is instantiated (infinitely often) at regularly spaced intervals. Each thread in $Poll^*$ waits indefinitely for a reply.

Components and Component Placement with ORC In this subsection an outline of a software component is given. In particular, those aspects of a component that are grid oriented are modelled. Traditionally, a software component may comprise functionality, input data, an interface (defining software dependencies) and an output file (hereafter called an output component). In a grid setting it is additionally necessary to supply *constraints* (which restrict the kinds of hardware that can be used to execute a component). In this paper two different kinds of constraints are defined: *minimum constraints*, which can be used to determine if a grid site offers appropriate resources; and *value constraints*, which can be used to rank grid sites according to their suitability for executing a given component. A component is defined to be a sextuple comprising an external interface (a set of component names, E), an output component name (o), functionality (f), data (d), a minimum constraint (a predicate MC) and a value constraint (an expression VC).

$$component = (E \ , \ o \ , \ f \ , \ d \ , \ MC \ , \ VC)$$

The exernal interface of a component c defines its dependencies such as input data sources and utilised service components. Functionality may be supplied as a combination of program code and service invocations.

Let us give an example. Consider a user who wishes to execute a FORTRAN program, F, on the grid using local data, d, and a data file, I, supplied by a third party, to produce an output component, R. The user may construct a data component, *indata*, and a program component, FP:

$$\begin{aligned} \mathit{indata} &== (\_, \_, \_, I, \_, \_) \\ FP &== (\{\mathit{Fortran90Compiler},\ \mathit{indata}\}, R, F, d, MC, VC) \end{aligned}$$

where MC and VC are constraints (see below). The components FP and *indata* may be sent to, and executed on, any site offering the service *Fortran90Compiler* and satisfying the hardware constraint MC.

Constraints Constraints are statements that can be used to specify hardware, performance, cost and network requirements. These have two forms: *minimum* constraints and *value* constraints. A minimum constraint is a set of requirements that *must* be satisfied by any site which is delegated to execute the associated component. A minimum constraint can be modelled abstractly as a predicate.

Let us give an example.The constraint that a grid machine should have at least 10 processors and a communication link with bandwidth of at least 10^9 bytes/sec may be expressed as a predicate with free variables p and b:

$$MC = p \geq 10 \wedge b \geq 10^9$$

For a particular site the parameters p and b may be instantiated, allowing the predicate to be evaluated. In order to evaluate a constraint MC on a site s it is necessary to acquire the values of all free variables by means of a dialogue. For example, p might be instantiated for a site, s, by means of a call $s.processors$. The advantage of modelling constraints as predicates is that (uninteresting) underlying dialogue can be excluded. Let s_1 be a site with $p = 4$ and $b = 10^6$ and s_2 be a site with $p = 16$ and $b = 10^9$. Let $MC(s)$ be the value found by instantiating the free variables of MC with the site parameters of s. Then $MC(s_2)$ but $\neg MC(s_1)$. However, consider the weaker constraint MC':

$$MC' = p \geq 4 \wedge b \geq 10^6$$

Now $MC'(s_1)$ holds (that is, a component with constraint MC' can be executed on s_1 as well as s_2). Note that $\forall g.MC(g) \Rightarrow MC'(g)$ – that is, use of a weaker constraint extends the set of possible sites for placing a component.

A minimum constraint provides a means of deciding whether or not a site can execute a component. However, it is useful also to be able to determine the"best" site on which to execute a component. A value constraint is an expression with free variables. The expression can be instantiated via a dialogue as above. Instantiation allows a set of potential sites to be ranked according to their ability to meet the value constraint.

Let us give an example. Consider a component, c, with an associated constraint

$$VC = 10^{10}/ct + b$$

Here ct is a unit computational cost and b is the available bandwidth. Suppose that the unit costs for s_1 and s_2 are 1 and 1000 respectively. Then $VC(s_1) > VC(s_2)$ – in this case a cheap but less powerful site is preferred.

As example of the usability of the approach, we consider the *site selection* problem. Suppose that a user (or manager) knows the constituent sites in a grid $\mathcal{G}$:

$$\mathcal{G} = \{g_1, \ldots, g_n\}$$

Consider the selection of an appropriate site on which to place a component, c, for execution.

Arbitrary selection: Grid site selection can be made by choosing the first suitable site to respond:

$$Select\_First(c, \mathcal{G}) = \{let(g) \text{ where } (g, v) :\in (\mid\!\!_{g_i \in \mathcal{G}} \ g_i.can\_execute(c))\}$$

This selection generates a call to each of the sites in $\mathcal{G}$ to determine which are suitable for the placement of c; the first site to respond returns its grid site name, g, and the value, $VC(g)$. Evaluation of $Select\_First$ publishes g. Note that if none of the sites can execute c then the evaluation will not terminate.

Selection using a site ranking operation. The first site to respond may not be the best location for placing c. The set of best sites on which to place c, as determined by the value constraint VC, is:

$$\{b \in \mathcal{G} \mid \forall g \in \mathcal{G}.VC(b) \geq VC(g)\}$$

Here all sites which maximise the constraint VC are included in the set. The ranking of a particular site, s, may be found using the operation call $s.can\_execute(c)$. The result of such a call is a pair comprising the site name and its constraint value; these results are passed to a local variable z by means of the operation $pass$. The expression *Rank*, below, constructs multiple threads which send a stream of site information to variable z; after calling $pass$ each thread is terminated.

$$\begin{aligned} Rank(c, \mathcal{G}) = z.null \gg \\ (\mid\!\!_{g_i \in \mathcal{G}} \ g_i.can\_execute(c) > (g, v) > z.pass(g, v) \gg 0) \end{aligned}$$

Here z is initialised by means of the site call $null$. The best site available after t time units may be selected as follows:

$$\begin{aligned} Top\_Rank(c, \mathcal{G}, t) = let(g) \text{ where} \\ (g, v) :\in (Rank(c, \mathcal{G}) \mid Rtimer(t) \gg z.highest) \end{aligned}$$

The pair which has the highest constraint value may be retrieved by the call $z.highest$. Evaluation of Top_Rank is guaranteed to terminate; however, when none of the active sites in $\mathcal{G}$ can execute c, a null site name is published.

Variants of this operation may be applied to rank sites using different metrics. For example, suppose that yellow pages directories provide site reliability information (the probability of a site responding, say). Then a ranking operation could be constructed using reliability information.

Enlarged selection using a trawling operation. A user who polls the set of sites $\mathcal{G}$ and does not receive in reply a valid site name may wish either to poll a larger set of sites or alter the given constraint set. A larger set of potential sites may be found using grid information directories.

$$\begin{aligned} Trawl(c, \mathcal{G}) = x.empty \gg \\ (\mid\!\!_{g_i \in \mathcal{G}} \ \ g_i.all\_can\_execute(c) > \mathcal{G}' > x.union(\mathcal{G}') \gg 0) \end{aligned}$$

Here each thread in $Trawl$ determines a set of sites having the potential to execute c. The sets are combined by distributed union and the result is held in a local variable, x. Distributed union is realised through a stream of *local* site calls, $union$, each of which has the side effect of extending x with $\mathcal{G}'$. The site call $empty$ is used

to initialise x. After t time units the set currently stored by x is extracted using the operation *get*:

$$Trawl\_Select(c, \mathcal{G}, t) = Top\_Rank(c, \mathcal{G}', t)$$
$$\text{where } \mathcal{G}' :\in (Trawl(c, \mathcal{G}) \,|\, Rtimer(t) \gg x.get)$$

Note that the set of sites $\mathcal{G}'$ found by browsing information directories may not be valid. Direct contact is made with each of these sites using the expression $Top\_Rank(c, \mathcal{G}', t)$ to verify that it can be used to place c.

Selection with weakened value constraints. A user may, in the event of not being able to find a suitable site to place c, weaken the associated component value constraint. Let c' be a component which is the same as c except that it has a weakened value constraint. A selection mechanism which first tries to find where to place c and, if unsuccessful, then tries to find where to place c' is:

$$Weakened\_Select(c, \mathcal{G}, t) =$$
$$(\text{if } \neg\mathsf{null}(g) \gg let(g) \,|\, \text{if } \mathsf{null}(g) \gg Trawl\_Select(c', \mathcal{G}, t))$$
$$\text{where } (g, v) :\in Trawl\_Select(c, \mathcal{G}, t)$$

This strategy can be repeated to allow the constraint to be further weakened.

It may happen that a user tries to select a site to place a component when the grid network is congested – in such circumstances responses to site calls may not be delivered. To make site selection more robust, it may be desirable to use a form of repeated polling.

4.1.2 Programming model institute perspective

To date, involvement in the Programming model Institute of CoreGRID has acted as a spur for QUB and UPC to pursue the modelling of grid systems. An abstract grid model has been proposed, with the ultimate aim of offering a grid application programmer a discipline within which to develop a grid program by first creating an ORC model and then refining this model, ideally through a series of correctness-assured steps, to an executable program. The importance of developing abstract models of grid applications, with the attendant benefits of clarity and amenability to formal reasoning, has been recognised in the Programming model Institute RoadMap (D.PM.03): 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"*.

It is anticipated that ORC will provide a means to specify the adaptive/autonomic aspects of grid applications, whether via the component managers of ASSIST (3.1) or the controllers of the GCM.

Currently four threads of activity are under way:

- A probabilistic reasoning framework is being developed which may be used to investigate and quantify reliability properties of grid networks [63].
- A Researcher Exchange Programme between QUB and UNIPI has commenced at the beginning of September 2006 with the aim of combining the abstract QUB/UPC grid model with the practical grid experience of UNIPI to investigate the modelling and derivation of grid systems. This work will be carried out in the context of the ASSIST programming environment of section 3.1.

- A PhD student at QUB is beginning investigation of the derivation of grid applications from ORC specifications. Initially, this work will target the Belfast e-Science Centre grid but will, it is anticipated, ultimately target the UK e-Science grid and the CoreGRID testbed.
- A Masters student at QUB is beginning investigation of the derivation of industrial related grid applications from ORC specifications. This work is being carried out in the context of the Belfast e-Science Centre (BeSC) where a number of industrial sponsored grid applications have been completed. Resources being available, it is intended to build on the work of BeSC to broaden experimentation with grid application modelling in industrial settings.

4.2 Deductive verification tools

Temporal logic, which was originally developed as a logical framework in which to describe tense in natural languages, is now considered to be an essential tool in both Artificial Intelligence and Computer Science. Temporal logic becomes crucial when the structures to be described invoke temporal aspects. This is relevant in the case of temporal databases, planning, program specification and verification, problem solving and information retrieval. For example, the sequence of steps of a program execution can be considered as a sequence of *moments*, or *states*, within a temporal logic. Thus, proofs about the correctness of programs correspond to proofs within an appropriate temporal logic [40]. A particular area in which temporal logics have been extensively used is in the specification and verification of properties of concurrent and distributed systems. The power of the temporal language used in formal specification of such systems allows the representation of a variety of complex properties relating to these systems, such as liveness, deadlock and invariance.

As the applications that require temporal reasoning become more refined, so the corresponding logical tools have to be extended. If a temporal model aims to represent the behaviour of a complex dynamic system, for example a complex multi-process system, the ability to refer to a range of possible execution paths in a model becomes important [28]. An appropriate logical framework to reason about such systems is called *branching-time* temporal logic. Here, the underlying model of time represents a choice of possibilities branching into the future. The first branching-time logics were originally developed for the specification of concurrent and distributed systems [28]. Varieties of branching-time logics are characterised by specific syntactic restrictions which, in turn, lead to different levels of expressiveness. Within these constraints, Computation Tree Logic (CTL) has been shown to play a significant role in many potential applications.

In our version of the syntax of CTL we use **start** as a constant that is true at the initial state of the tree, operator $\bigcirc$ stands for 'next time' in the future, $\Box$ stands for 'always in the future' and $\Diamond$ stands for 'at some time in the future', while path quantifiers **A** and **E** stand for the universal and existential path quantifiers respectively.

It has been observed that this logic, first proposed in [28], is sufficient to express most of the properties concerning branching-time models, that is properties of simple concurrent programs (which do not deal with fairness). There are several extensions of CTL, for example, CTL^+, ECTL, ECTL^+, and $\text{CTL}^\star$. In particular, CTL can be generalized to Extended CTL (ECTL) [40], which incorporates simple fairness constraints. It has been shown that CTL and ECTL can be respectively extended to CTL^+ and ECTL^+, where Boolean combinations of temporal modalities are allowed. Here, CTL^+ is of equivalent expressive power to CTL, while ECTL^+ is strictly more expressive than ECTL [40]. CTL^* is commonly considered as a full branching-time logic [42] as it is the most expressive logic of this family.

With most research in verification methods associated with the specification languages of CTL–type logics being concentrated on developing the *explorative approach*, relatively little attention has been paid to their efficient decision procedures, and potentially more efficient, proof methods. Clausal resolution is one of the powerful methods widely used in the framework of classical logic [8]. In [43] a clausal resolution approach for the propositional linear-time temporal logic (PLTL) was proposed. Its methodology suggests a definition of three key components: a translation to a *normal form*, an application of *step*, or classical-style resolution, and a novel *temporal* resolution technique. It has also found an efficient implementation [32] and has been further refined [33, 44].

This technique of generating a normal form based on fixpoint characterization of temporal operators [24, 40] has given a significant methodological tool which has made it possible

In [19, 20] a clausal resolution approach to CTL was developed, extending the original definition of the method for the linear-time case [43]. It has been further extended to capture richer formalisms, ECTL and ECTL$^+$ [23].

Given a formal specification of a distributed system there are two major approaches to formal verification of this specification: explorative and deductive. While the former approach is fully automated, as in the case of model checking, its application, in general, is restricted to finite state systems. On the other hand, methods of the second, deductive approach, can handle arbitrary systems providing uniform proofs.

The general problem structure that we are trying to solve is given in Figure 5. Here we are attempting to analyze a system specification (S) and provide some (formal) verification of its properties (V).

If the specification (S) is given in a high-level language, for example a logic, then we can either translate (S) to an automaton (A) and then carry out as verification (V) an automaton emptiness check, or we can carry out the translation from (S) to a Normal Form (N) and then use as verification (V) some form of efficient deduction, for example clausal resolution. The complexity of the translation from (S) to an automaton (A) is usually exponential [67]. For example, a PLTL formula ϕ can be translated into a Büchi automaton with $2^{O(|\phi|)}$ states [67]. In contrast, checking non-emptiness for a Büchi automaton is decidable in linear time [41, 67]. Note, however, that the situation is different when we carry out the translation of a PLTL formula ϕ into alternating automata [67, 17], where the translation (S)$\rightarrow$(A) results in an automaton of the size $O(|\phi|)$, while checking non-emptiness involves an exponential blow-up. No direct methods of checking non-emptiness of alternating automata are known. Usually an alternating automaton is simulated by a standard non-deterministic automaton and the known emptiness check is applied to the latter [17, 67].

On the other hand, the complexity of the translation from (S) to (N) is polynomial or often linear [45]. In contrast, verification of formulae in the Normal Form ((N) $\rightarrow$ (V)) is usually exponential since it involves some form of proof (in our case, clausal resolution) [15]. However, we are often able to use either improved proof strategies [44] or restricted forms of the normal form [34] in order to improve the practical efficiency of such proof.

Our particular concern here was the relationship between the Normal Form (N) given as the normal form for PLTL or CTL and the Automaton (A) in the diagram above. It has been shown that a normal form for the formulae of PLTL can represent Büchi automata [21]. The essential reason for this is that, in translating a problem specification into our normal form, we actually derive clauses within a fragment of quantified propositional linear-time temporal logic (QPLTL) [62, 51]. In particular, formulae within $\mathrm{SNF}_{\mathrm{PLTL}}$ are existentially quantified. In order to utilise the normal form as part of a proof, we effectively skolemize the normal form producing temporal

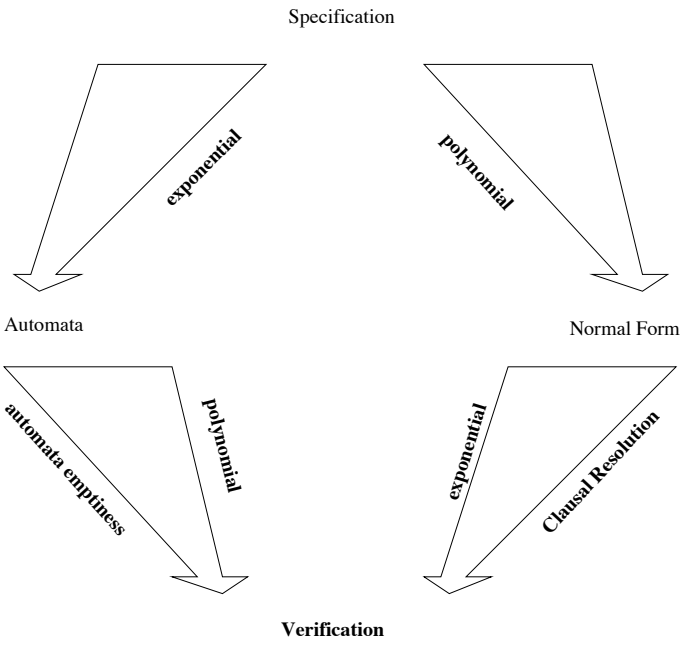

Figure 5: Specification-Verification Problem

formulae without any quantification (i.e. PLTL).

Having established this relationship between the normal form and Büchi automata, we are able to represent a problem specification directly as a set of formulae in the Normal Form and apply a resolution based verification technique to the latter. With the set of works showing that normal form initially developed for CTL, can also capture logics ECTL and ECTL$^+$, we can expect similar results in the branching-time setting. This enables us to directly specify properties of a grid system in the language of normal form with the subsequent application of the resolution technique as a deductive verification tool.

The verification of the generated specification includes application of the two types of resolution rules already defined in [18, 20]: *step* resolution (SRES) and *temporal* resolution (TRES).

Step Resolution Rules. Step resolution is used between Formulae that refer to the *same* initial moment of time or *same* next moment along some or all paths. In the formulation of the SRES rules below l is a literal and C and D are disjunctions

of literals. Two step resolution rules that will be used in our example are given below.

SRES 1

$$\frac{\mathbf{start} \;\Rightarrow\; C \vee l \qquad \mathbf{start} \;\Rightarrow\; D \vee \neg l}{\mathbf{start} \Rightarrow C \vee D}$$

SRES 3

$$\frac{P \Rightarrow \mathbf{A}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{E}\bigcirc(D \vee \neg l)_{\langle \mathsf{ind} \rangle}}{(P \wedge Q) \Rightarrow E\bigcirc(C \vee D)_{\langle \mathsf{ind} \rangle}}$$

Temporal Resolution Rules. The temporal resolution rules resolve the so called A and E loops in some property l [19], i.e. the situations where, given that P is satisfied at some point of time, l occurs always from that point on all or some path respectively, with the eventuality $\neg l$.

When an empty constraint is generated on the right hand side of the conclusion of the resolution rule, we introduce a constant **false** to indicate this situation and, for example, the conclusion of the SRES 1 rule, when resolving $\mathbf{start} \;\Rightarrow\; l$ and $\mathbf{start} \;\Rightarrow\; \neg l$, will be $\mathbf{start} \;\Rightarrow\; \mathbf{false}$, which terminates the verification.

4.2.1 Components and grids and deductive verification

We worked on providing the deductive verification technique for the components described in a Fractal framework. The developments described in the previous section allow us to reason about the configuration/(re)configuration protocols of a Grid component model on behalf of the following formal framework:

$$\boxed{\text{FCM}} \longrightarrow \boxed{SNF_{ctl}(FCM)} \longrightarrow \boxed{\text{BTR}}$$

Figure 6: Deductive verification of Fractal Model

Here we suggest the translation of the Fractal component model (FCM) into the SNF_{CTL}(FCM), the SNF_{CTL} based formal specification of FCM, and to apply the 'branching temporal resolution' method (BTR), the temporal resolution technique defined over the set of SNF_{CTL} clauses.

We have shown how to extract the desired SNF_{CTL} based temporal specification for a given component model [14]. The output system would have an intelligent verification engine strengthened with the corresponding search techniques [22] as well as with the possibility of invoking powerful refinement methods developed for the resolution in the classical setting [8].

In general, the initial configuration of a Fractal component is given by the description of the component using Fractal ADL.

From this first state, reconfiguration is obtained by triggering appropriate actions on the life-cycle, the binding, and the content control interfaces. A reconfiguration can be triggered by any component that has a reference to a correct non-functional interface.

We will illustrate our approach by considering a simple printing queue component model which consists of a client and one printing queue component as primitives. The client interfaces are of type CI_a and the server interfaces of the printing queue are of type SI_r. Finally, we have a simplified version of a life-cycle controller that allows to safely add or remove a binding between a client and the printing queue.

We will only specify the safe-unbinding part of a reduced Life-Cycle Controller (LCC) so that it can be used in the deductive reasoning. Note that it is always possible to create new controllers if needed, in this case an appropriate set of formal specifications for each controller must be provided using a similar procedure. If a controller follows the standard Fractal model, a standard set of general temporal

logic rules can be called and then modified to match the specification; otherwise, in the case of user-made definitions, the programmers themselves must provide the rules matching the criteria followed in the creation of the definition.

Next we will let the propositions $Bound_1, \ldots, Bound_n$ denote the bindings between components. The format that each may take is $Bound_i(CI_a, SI_r)$ $(1 \leq i \leq n)$ which is a proposition that (when true) specifies that a component with Client Interface CI_a is bound to the Server Interface SI_r. In this example we have two primitive components, one for the Printing Queue and one for the Client using the Printing Queue. We would add as many of these propositions as necessary to describe the system.

LCC is a proposition which when true signifies that the Life Cycle Controller is active.

Before introducing the Life Cycle Controller Formula we would need to specify how components are started and stopped. However, for illustration in the context of this paper we will only provide a partial specification of the Life Cycle Controller and two primitive components; we only deal with the formula that captures the bindings of the two components. We will model the start of the components by attaching them to **start** .

Now we introduce the formula for our version of the Simplified Life-Cycle Controller:

$$\neg LCC \wedge \neg(Bound_1(CI_a, SI_r) \vee Bound_2(CI_a, SI_r))$$
$$\Rightarrow \mathbf{A}\,\Box LCC \Rightarrow (Bound_1(CI_a, SI_r) \wedge Bound_2(CI_a, SI_r))$$

which states that if neither of the components are bound and the LCC is not active then in all possible computations when the LCC is active then we must have the two components bound.

In the following example the Client can send a request for printing: $req(CI_a)$ abbreviated below as req. When true, this proposition states that a printing request has been raised by the client which possesses the client interface CI_a. Similarly $print$ is a proposition stating that a printing request has been satisfied by the printer.

Let us consider a simple printing queue component model (see figure 7) which consists of one client and one printing queue component as primitives. The client interface of the client is labeled CI_a, and the server interfaces of the printing queue is labeled SI_r. We will also consider a simplified life-cycle controller LCC that allows us to safely remove a binding between a client and the printing queue. This simple example is sufficient to demonstrate the potential of deductive reasoning, applied to a fractal model.

We will take into consideration the safety part of the specification and its requirements. The Life-Cycle Controller LCC does not have a set specification being a non-functional component. We suggest that the system has a common protocol of communication (both Client and Printing Queue must follow a common process when a request is raised).

Client specification:

(1) $req \Rightarrow$ $\mathbf{A}(req\,\mathcal{U}\,(req \wedge print))$	Request is kept until it is possible to execute it
(2) $req \Rightarrow \mathbf{A}(\neg req2\,\mathcal{U}\,print)$	There will be no other request until job is printed
(3) $req \Rightarrow \mathbf{A}\Diamond\neg req$	The request for print will be eventually released

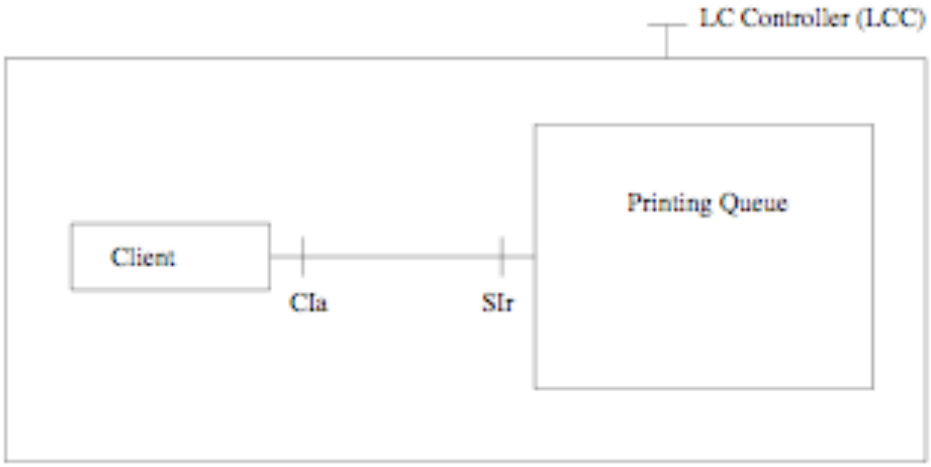

Figure 7: Example in Fractal

The complete specification of the primitive:

$$\textbf{start} \Rightarrow \neg req \wedge (1) \wedge (2) \wedge (3)$$

where $\neg req$ defines the initial state for Client primitive.

Printing queue specification:

(4)$\mathbf{A}\,\Box\neg(print \wedge print2)$	Mutual Exclusion property: at every point in time, the printer can perform at most one printing operation:
(5) $\mathbf{A}(\neg print\, \mathcal{W}\, req)$	There is no printing unless requested
(6) $print \Rightarrow \mathbf{A}\Diamond\neg print$	Printing will eventually end
(7) $req \Rightarrow \mathbf{A}\Diamond print$	The request for printing should be granted

The complete specification of the primitive:

$$\textbf{start} \Rightarrow \neg print \wedge (4) \wedge (5) \wedge (6) \wedge (7)$$

Finally we specify the Life-Cycle Controller properties which affect the receiving of a printing request and the printing itself:

$$\textbf{start} \Rightarrow [(\neg LCC \wedge \neg(req \vee print)) \Rightarrow \mathbf{A}\,\Box(LCC \Rightarrow (req \wedge print))]$$

When the life-cycle controller is activated, it ensures that Client Interface and Server Interface are bound, therefore allowing for requests to be sent from the Client, and prints to be carried out by the Printing Queue, for the specific binding.

To apply deductive reasoning to this model, various properties could be taken into consideration. To illustrate the approach, let us consider a relatively simple example. Let p stands for $\neg req(CI_a, SI_r) \wedge \neg print(CI_a, SI_r)$. Assume now that during the reconfiguration of the system the following property should be verified:

$$\ddagger \qquad A(\,\Box\Diamond p \wedge \Diamond\,\Box\neg p)$$

In the next section we will show how this formula can be represented in terms of $\mathrm{SNF}_{\mathrm{CTL}}$ and then apply to this specification the resolution technique as a verification method.

To verify (‡) we apply the resolution method to the set of SNF_{CTL}(‡) deriving an empty clause [14]. This means that the refutation procedure has ended with the contradiction, hence ‡ itself is unsatisfiable (and its negation valid).

4.2.2 Programming model Institute perspective

The University of Westminster team worked on the issues of deductive verification complying with one of the tasks 3.2 and 3.2 of the Roadmap. In particular, we have aimed to establish the theoretical foundations for dynamic configuration and reconfiguration of components in a distributed system in an automated way based on the mathematically justified GCM. Following the initial plan, firstly, the team has started working with the definition of configuration and reconfiguration and adaptation of branching-time specification techniques to GCM. Our approach has been successfully applied to simple GCM. We have found that the proposed specification langauge may need to be extended to achieve a higher level of expressiveness and to enable verification.

Secondly, the team has concentrated on the application of formal verification techniques to the obtained specifications. We have applied resolution based verification techniques to a simple component model whose specification follows the Fractal framework. This should be extended in the future to cover more sophisticated cases. Ideally, we are hoping to consider a "real-life" case taken from some working system.

A PhD student at the UoW has been working on adapting deductive verification techniques to the Fractal component model. One of the important tasks derived from our experience which the student is currently investigating is an attempt to define and extract Metadata needed for the formal specification. Observe, that the extraction of metadata is itself a very important task that is related to many activities of the Programming Model institute.

A close collaboration with INRIA has been established and will have to continue to enable a correspondence of our developments and Fractal GSM as well as to provide life examples for the formal specification and verification to test.

During the reported period a few common issues have been found between our work and the collaborative work of QUB and UPC on ORC, the language for specifying the orchestration of distributed services. Thus, further steps towards closer collaboration, comparison of our approaches looks very promising.

Finally, our work on the deductive verification was carried out as part of AUTOGRID, the project started in the UoW. The project aims at the development of theoretical foundations of a reasoning engine for multi-layer self-organizing systems, which will automatically manage reconfiguration of components in a system in a safe and optimal way. In [22] we proposed a framework that enables the application of the Inferential Erotetic Logic (IEL) tools aiming at optimisation of the process of reconfiguration of a component model.

4.3 Specification and verification of component behaviour

Component programming proposes to split the system into smaller pieces of software that interact through well defined frontiers, called interfaces. In hierarchical component models, and in particular in the Grid component model that is being defined by the CoreGRID Institute, new components (called composites) can be created by composing existing (and smaller) components in a hierarchical fashion, enhancing the reusability of component libraries.

Components can be bound through their interfaces only if typing requirements are met, ensuring a basic compatibility. Although the GCM clearly exposes both provided and required services (e.g. method signatures) of each component through its interfaces, it is well-known that static typing of bound interfaces is not enough for guaranteeing a correct assembly of components: even if they statically match their interfaces, off-the-shelf components may not work together due to the lack of a dynamic behavioural compatibility, resulting in mismatch between their communication protocols, and often deadlocks.

Our work aims at providing methods and tools for expressing the behaviour expected by software components, help the component developer ensure that his code matches the specification, and help the component user check if assemblies behave properly.

Expressing a precise behaviour requires a solid mathematical background. Formal methods aim at defining a system without ambiguity, so that formal tools, implementing either theorem proving or model-checking, can be used to bring strong guarantees on the program properties. For being usable in real-life programming environments, these methods require support from automatic software tools which hide the complexity of the underlying logics from the developer. Although we can find successful cases in the hardware industry, the usage of formal methods and of verification tools is still limited when it comes to software.

Compared to existing formal methods in classical (sequential) software development, the context of distributed components is more complex because of the intricate interaction between the distributed parts of an application. This requires specific care in handling asynchrony, remote failures, communication delays, etc. We also have to take into account the managment functionalities provided by many component frameworks: deployment, life-cycle management, component updates, or even dynamic changes in the topology. On the other hand, component frameworks provide both a programming model that helps the developer to abstract away from the details of the underlying execution platform, and tools for supporting these abstractions (description languages, middle-ware layers providing strong behavioural guarantees, etc.).

In the Oasis team we have been developing a semantic model allowing the description of parameterized structured applications communicating through synchronous or asynchronous messages, and have applied this model to various kinds of hierarchical components, including distributed and Grid components. We are now in the process of providing prototype tools supporting this approach.

Amongst the research work being done on the formalisation of distributed component behaviour, the closest to the motivations expressed above is certainly the one developed by the Sofa team [60], similar to the Fractal component model. In Sofa, components have a *frame* (specification) and an *architecture* (implementation) protocols, and verification is done through a trace language inclusion of the *architecture* within the target *frame*. In a different flavor, the work of Carrez et al on behavioural typing of components [27] defines a *sound assembly* and compatibility concepts which ensure correctness of the composition, but in a framework based on the Corba component model CCM [57], where they have no hierarchy of components.

In the same spirit, we must mention the work of the ArchWare project [54, 58], even if it does not explicitly address components. ArchWare was an IST-5 european project, in which was defined a family of architecture specification languages based on a π-calculus like semantics, including an Architecture Description Language π-ADL, a logical requirement language π-AAL, etc. Being explicitly based on software architecture, these formalisms are close to our concerns, except for the absence of component management functionalities.

There exist a number of verification platforms for process algebras, and more

recently some supporting components. The only one supporting behaviours of hierarchical components that we are aware of is Sofa; it includes a static analysis module based on the Java Path Finder tool, and a home-made model-checker implementing the Sofa compliance relation. The main difference with our approach is that, being based on a trace language semantics, they have no support for (congruent) minimization of state space. The ArchWare people also propose a set of verification tools, using the CADP toolset, and taking into account only a finitary subset of their π-calculus based model.

We made slightly different choices: our models are based on bisimulation theories, and we take advantage of their congruence properties, together with the structuring capability of hierarchical components, to keep their state-space complexity manageable. Our work has been concentrated in the automatic building of behavioural models for distributed component systems. In [10] we introduced a new semantic model named pNet extending the networks of communicating automata [7], by adding parameters to their communication events and processes in the spirit of symbolic transition graphs [53]. In [12] we used pNets to model the behaviour of Fractal (synchronous) components, including the representation of the Fractal "non-functional operations" for the dynamic management of the component assemblies. Then in [13] we extended this work to the distributed implementation of Fractal using the ProActive library, which features asynchronous communication between distributed components. Finally in [11] we describe the tools we have developed suppporting the construction of pNet models from the description of the components' architecture, their instantiation to finite models and interface with efficient model-checking tools, and we illustrate their use in a non-trivial case-study.

We will describe these methods in detail in the next section.

4.3.1 Specification and verification of GRID component programs

Our pNet model is a very expressive tool for representing the behavioural semantics of realistic distributed systems, that allows us to formalise in a coherent framework the whole chain from behaviour specification and expression of requirements, to the intermediate formats required for using automated model checking engines.

We can use it as a rudimentary language for the specification of primitive component behaviours (in [11] we use Lotos as a syntax for representing parameterized labelled transtion systems (pLTS), that are the leaves of a pNet hierarchy), though we would prefer a much higher level language to be exposed to the component system developer. We use it also as the target of code analysis in [10], so we have the possibility to compare the semantics extracted from static analysis of a primitive component with its specification (the tools for this are not yet available).

The specification of component architecture is based on the Architecture Description Languages naturally available in the component frameworks. Historically we started with the Fractal ADL, and proposed an extension to the Fractal community, that encompasses several behaviour specification languages [26]. The same extension is also being defined in CoreGRID WP3 D.PM.04. The extension consists in attaching to each level of the architecture (i.e. to primitives and composites components) a behaviour description in some dedicated behaviour language (including Sofa behaviour protocols and our pLTS in Lotos syntax). Because they require an external parser, and may be quite large, the behaviours are usually provided in separate files. An important technical point is that the information from the ADL itself is not sufficient for expressing the behaviour events, we also need access to the specification of the component interface, namely the methods and their argument types used in communication.

The ADL2N tool (developed by the Oasis team), described in [11], generates the pNets modelling a component hierarchy, expressing the synchronisation constraints

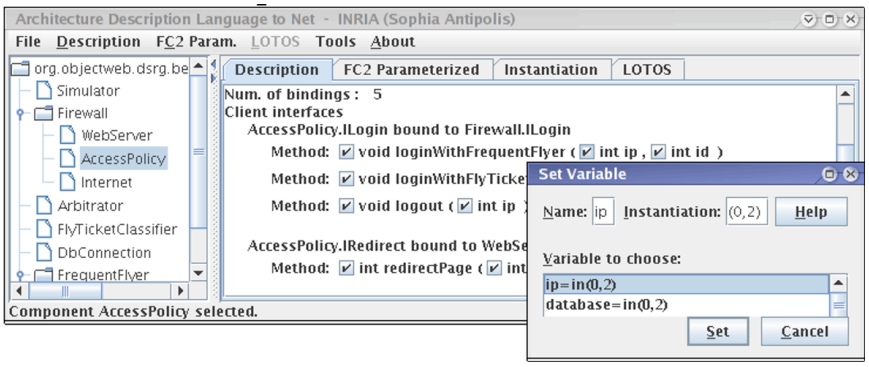

Figure 8: The ADL2N tool

corresponding to component bindings. The tool also generates the controllers required for modelling the life-cycle and binding management of the components. ADL2N also provides means to specify precisely which internal events should be observed through the hierarchy, allowing an early control of the well-known state-explosion problem.

The next step in the verification activity is the definition of abstractions transforming our parametrized models into finite-state structures acceptable by the model-checking engines. The types authorized by our methodology for the arguments of communication events are simple (first-order) types. We use finite partitions of those types to define abstract interpretations that preserve safety properties of the models. We call "instantiation" the abstraction of a pNet into a finitary Net structure, and apply the same abstraction to the Lotos models of the primitive components. Then we build a finite state input to the verification engines from the CADP toolset [47]; CADP provides us efficient tools for building, reducing, and model-checking our behaviour models, including distributed and on-the-fly techniques.

We have shown how to use automatic model-checking tools to check requirements, in the form of parameterized temporal logic formulas, against the generated pNet models. Of course the instantiation at the previous step must be consistent with the specific values, operators and predicates occurring in the formulas to be proven.

To illustrate our results, let us give the main ideas from the case-study in [11]. The example comes from a realistic case-study provided by France Telecom, previously analysed by the Sofa team in Prague [59]. This example has several qualities that motivated its choice: first it is extracted from a realistic case-study, that will hopefully be part of a standard set of examples provided to the whole Fractal community for comparisons. Secondly it is small enough for a description in a research paper, but big enough to illustrate some scalability issues. Last it is suitable to illustrate some of the features we wanted to show, including the treatment of parameters, and the modelling of multicast interfaces.

The case study concentrates on a subset of an airport wifi system, involving airline databases, a firewall and a webserver, and users with various status allowing access to the wifi network. From the ADL description of the component system, extended with Lotos specification of each primitive component, we build a parameterized model of various subsets of the system. Then we use small instantiations of the parameters (web urls, ticket ids, ticket validity, databases) to check deadlock detection and CTL properties. During this process the state-space is built in a compositional manner, using hiding and minimisation as much as possible; the biggest intermediate structure, for a model including only the functional behaviour of the

example, has only 5000 states. Indeed we are able to find a deadlock in our specification of the system, and to disprove a CTL formula (expressing the inevitability of an event), and show how the model-checker gives us diagnostics.

4.3.2 Programming model Institute perspective

The previous results are general enough to apply to various kinds of programming methodologies, from module-based or object-oriented languages, and to various kinds of behavioural semantics, including both synchronous and asynchronous applications. Indeed, the results in [10] were established in the framework of distributed, asynchronous applications based on active objects.

The extension of our work to hierarchical components was initiated in the context of the Fractal component model. It was an important step: the separation from the code of primitive components (functional code) and the architecture description (Fractal ADL) gave us the opportunity to attach a behaviour specification to each level of the component structure. This is the key to safe reusability of components: beyond static typing of interfaces, it allows one to guarantee that a composition of components behaves well together, respects their respective protocols, does not block on deadlocks, achieves progress, and more generally implements their own behavioural specification. At the same time it brought more complexity, because Fractal components are more than modules, and come with their life-cycle controllers, and provisions for dynamic updating of the architecture.

There are two other steps before reaching grid components: dealing with asynchronous communication between distributed components, and dealing with "group communication", or more precisely with multicast and gathercast policies for the interaction between large numbers of similar components in grid computing applications. In [9] we showed how to model the request queues and proxies of the distributed implementation of Fractal based on the ProActive library. The implementation in our model generation tools is not yet available, and will require specific care for dealing with finite abstractions of the queues. In [11] we also give an example including multicast / gathercast modeling, that shows that it is possible on small configurations to keep the state explosion local within the pNets structure.

A possible manner in which to consider this work is the following: having a strong formal basis, the GCM implementation using the ProActive library provides a number of guarantees to the Grid application developer, namely safety of the message mechanism, transparency of distribution, determinacy of distributed computation. But it does not prevent programmers from creating complex communication and distribution patterns, that may lead to inadequate usage of components, deadlocks in communications or in computation dependencies, or non-termination. Our tools cannot provide a 100% safe answer to these questions (this is non-decidable), but provide a reasonable compromise between the capacity of automatic model-checking engines and the effort required for specification and validation.

5 Conclusions

We have presented in this report the more relevant research activities that the partners of the Programming model Institute have performed in the framework of the Institute Task 3.3: "advanced programming models". Part of the research results are related directly to the implementation of "advanced" actual grid programming models. These are discussed in Section 3. Another part of the results and research topics (those presented in Section 4) are related to abstract, theoretical models that can be used to investigate/design/model different properties of grid programming environments, in particular of advanced programming environments based on component technology concepts.

The research topics and results presented and discussed in this work are clearly placed in the framework of the research activities of the Programming model Institute, as stated in the Institute roadmaps D.PM.01 and D.PM.03. In particular, each of the six "technical" Sections in this document ends with a part discussing how the technical matter just presented relates to the activities of the Programming model Insitute and, in particular, how it contributes to the general Institute roadmap and to the specific Task 3.3 roadmap.

The activities described in this document must be considered as the basis on which the integration among the Institute partners will be built in the last two years of the NoE, in particular on the subjects listed under the Task 3.3 activities. Some (actually, most) of the research activities presented in this document have already led to well defined integrated research involving more than a single CoreGRID partner. Some of the ASSIST and HOC results have already been (or they are currently being) migrated to the GCM design, as well as some of the results achieved in Dynaco and also the theoretical framework described in Section 4.3. On the other hand, HOC has already led to specific joint activities between CoreGRID partners not previously working on integrated research issues (WWU Muenster and UDELFT on advanced programming model process scheduling), ASSIST led to the same result (UNIPI and INRIA/OASIS + INRIA/PARIS teams cooperating on autonomic component management, skeleton programming models on top of components, shared memory support for component grid computations) and the ORC framework stimulated previously unforeseen common research (involving QUB, UPC and, more recently, UNIPI and, in part UoW). Overall, these provide evidence that the durable integration which constitutes the main goal of CoreGRID is actually being built within the Programming model Institute and within its Task 3.3, in particular.

References

[1] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In *Future Generation Grids*, CoreGRID series. Springer, November 2005.

[2] Marco Aldinucci, Marco Danelutto, and Marco Vanneschi. Autonomic QoS in ASSIST grid-aware components. In *Proceedings of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, Montbéliard, France, February 2006. IEEE.

[3] Marco Aldinucci, Alessandro Petrocelli, Edoardo Pistoletti, Massimo Torquati, Marco Vanneschi, Luca Veraldi, and Corrado Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of 11th Intl. Euro-Par 2005: Parallel and Distributed Computing*, volume 3648 of *LNCS*. Springer Verlag, August 2005.

[4] Globus Alliance. Incubator project. http://dev.globus.org/wiki/Incubator/Incubator_Management.

[5] Apache Organization. The apache soap web site. http://ws.apache.org/soap/.

[6] Apache Organization. The Apache Web Services Project: Axis. http://ws.apache.org/axis.

[7] A. Arnold. Nivat's processes and their synchronization. *Theor. Comput. Sci.*, 281(1-2):31–36, 2002.

[8] L. Bachmair and H. Ganzinger. A theory of resolution. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 2. Elsevier, 2001.

[9] T. Barros. *Formal specification and verification of distributed component systems*. PhD thesis, Université de Nice - INRIA Sophia Antipolis, November 2005.

[10] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, volume LNCS 3235, Madrid, September 2004. Spinger Verlag.

[11] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components : The vercors platform. In *3rd workshop on Formal Aspects of Component Systems*, Prague, Tcheque Republic, Sep 2006.

[12] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In Patrice Godefroid, editor, *Model Checking Software, 12th International SPIN Workshop*, volume LNCS 3639, pages 154–168, San Francisco, CA, USA, August 2005. Springer.

[13] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[14] Alessandro Basso, Alexander Bolotov, Artie Basukoski, Vladimir Getov, Ludovic Henrio, and Mariusz Urbanski. Specification and verification of reconfiguration protocols in grid component systems. In *To be published in the Proceedings of IS-2006*, 2006.

[15] Artie Basukoski and Alexander Bolotov. Search strategies for resolution in ctl-type logics: Extension and complexity. In *TIME*, pages 195–197, 2005.

[16] Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*. Springer, 2003.

[17] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, California, June 1994. Springer-Verlag.

[18] A. Bolotov. *Clausal Resolution for Branching-Time Temporal Logic.* PhD thesis, Department of Computing and Mathematics, The Manchester Metropolitan University, 2000.

[19] A. Bolotov and C. Dixon. Resolution for branching time temporal logics: Applying the temporal resolution rule. In *Proceedings of the 7th International Conference on Temporal Representation and Reasoning (TIME2000)*, pages 163–172, Cape Breton, Nova Scotia, Canada, 2000. IEEE Computer Society.

[20] A. Bolotov and M. Fisher. A clausal resolution method for CTL branching time temporal logic. *Journal of experimental and theoretical artificial intelligence*, (11):77–93, 1999.

[21] A. Bolotov, M. Fisher, and C. Dixon. On the relationship between w-automata and temporal logic normal forms. In *Proceedings of the Advances in Modal Logic/International Conference on Temporal Logic 2000*, Leipzig, October 2000. Extended version accepted for publication in Journal of Logic and Computation.

[22] A. Bolotov, P. Lupkowski, and M. Urbanski. Search and check. problem solving by problem reduction. In *Proceedings of The 8th International Conference on Artificial Intelligence and Soft Computing*, pages 87–106, 2006.

[23] Alexander Bolotov and Artie Basukoski. A clausal resolution method for extended computation tree logic ectl. *J. Applied Logic*, 4(2):141–167, 2006.

[24] J. Bradffield. and C. Stirling. Modal logics and mu-calculi. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 293–330. Elsevier, North-Holland, 2001.

[25] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Afpac: Enforcing consistency during the adaptation of a parallel component. *Scalable Computing: Practice and Experience*, 7(3):83–95, September 2006. electronic journal (`http://www.scpe.org/`).

[26] A. Cansado, L. Henrio, and E. Madelaine. Towards real case component model-checking. In *5th Fractal Workshop*, Nantes, France, July 2006.

[27] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound assembly of components. In *Forte'03 conference*, number 2767 in LNCS, Berlin, 2003.

[28] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs. Proceedings of Workshop*, volume 131 of Lecture Notes in Computer Science, pages 52–71. Springer-Verlag, 1981.

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

[30] Murray I. Cole. *Algorithmic skeletons: a structured approach to the management of parallel computation*. MIT Press & Pitman, 1989.

[31] Universit de Nice Sophia Antipolis. The proactive web site. http://www-sop.inria.fr/oasis/ProActive/.

[32] C. Dixon. Search Strategies for Resolution in Temporal Logics. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 672–687, New Brunswick, New Jersey, July/August 1996. Springer-Verlag.

[33] C. Dixon. Temporal resolution using a breadth-first search algorithm. *Annals of Mathematics and Artificial Intelligence*, 22, 1998.

[34] C. Dixon, M. Fisher, and M. Reynolds. Execution and Proof in Horn-Clause Temporal Logic. In H. Barringer, M. Fisher, D. Gabbay, and G. Gough, editors, *Advances in Temporal Logic*, volume 16 of *Applied Logic Series*, pages 413–433. Kluwer, 2000. Proceedings the Second International Conference on Temporal Logic (ICTL).

[35] Catalin Dumitrescu, D.H.J. Epema, Jan Dünnweber, and Sergei Gorlatch. User Transparent Scheduling of Structured Parallel Applications in Grid Environments. Technical Report TR-0034, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, 2006.

[36] Jan Dünnweber, Anne Benoit, Murray Cole, and Sergei Gorlatch. Integrating MPI-skeletons with Web services. In *PARCO*, Advances in Parallel Computing, 2005.

[37] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A grid Service Architecture for Higher-Order Components. In *International Conference on Services Computing (SCC04), Shanghai, China*, pages 288–294, Washington, USA, 2004. IEEE computer.org.

[38] Jan Dünnweber, Sergei Gorlatch, Françoise Baude, Virginie Legrand, and Nikos Parlavantzas. Towards automatic creation of web services for grid component composition. In Vladimir Getov, editor, *Proceedings of the Grids@Work Plugtest, Sophia-Antipolis, France*, October 2005.

[39] Jan Dünnweber, Sergei Gorlatch, Sonia Campa, Marco Danelutto, and Marco Aldinucci. Using code parameters for component adaptations. In Sergei Gorlatch, editor, *Proceedings of the CoreGRID Integration Workshop, Pisa, Italy*, November 2005.

[40] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics.*, pages 996–1072. Elsevier, 1990.

[41] E. A. Emerson. Automated reasoning about reactive systems. In *Logics for Concurrency: Structures Versus Automata, Proc. of International Workshop*, volume 1043 of Lecture Notes in Computer Science, pages 41–101. Springer-Verlag, 1996.

[42] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. In *STOC 1984, Proceedings of*, pages 14–24, 1984.

[43] M. Fisher. A Resolution Method for Temporal Logic. In *Proc. of the XII International Joint Conference on Artificial Intelligence (IJCAI)*, pages 99–104, 1991.

[44] M. Fisher and C. Dixon. Guiding Clausal Temporal Resolution. In H. Barringer, M. Fisher, D. Gabbay, and G. Gough, editors, *Advances in Temporal Logic*, volume 16 of *Applied Logic Series*, pages 167–184. Kluwer, 2000. Proceedings the Second International Conference on Temporal Logic (ICTL).

[45] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic (TOCL)*, 1(2):12–56, 2001.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1995.

[47] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[48] Sergei Gorlatch and Jan Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

[49] Next Generation Grid Expert Group. NGG report 1-3. Reports 1 to 3 available at `http://cordis.europa.eu/ist/grids/pub-report.htm`, 2006.

[50] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[51] Y. Kesten and A. Pnueli. A complete deductive system for QPTL. In *Proceedings of the 10th Annual IEEE Symposium of Logic in Computer Science*, pages 2–12, 1995.

[52] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR*, pages 398–416, 1993.

[53] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, August 1996. LNCS 1119.

[54] R. Mateescu and F. Oquendo. π-AAL: An Architecture Analysis Language for Formally Specifying and Verifying Structural and Behavioural Properties of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 31(2):1–19, March 2006.

[55] J. Misra. Computation Orchestration: A basis for a wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, ASI. NATO, 2004. Marktoberdorf, Germany.

[56] OASIS Technical Committee. WSRF: The Web Service Resource Framework, http://www.oasis-open.org/committees/wsrf.

[57] OMG. Corba components, version 3. Document formal/02-06-65, June 2002.

[58] F. Oquendo. π-ADL: An Architecture Description Language based on the Higher Order Typed ?-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM Software Engineering Notes*, 29(3):15–28, May 2004.

[59] F. Plasil P. Jezek, J. Kofron. Model checking of component behavior specification: A real life experience. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[60] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

[61] WWU Münster PVS work group. The higher-order component web site. http://pvs.uni-muenster.de/pvs/forschung/hoc.

[62] A. Sistla. *Theoretical issues in the Desing and Verification of Distibuted Systems.* PhD thesis, Harvard University, 1983.

[63] A. Stewart, J. Gabarró, M. Clint, T. Harmer, P. Kilpatrick, and R. Perrott. Estimating the Reliability of Web and Grid Orchestrations, 2006. Presented at the *CoreGRID Integration Workshop 2006*, Poland.

[64] A. Stewart, J. Gabarró, M. Clint, T. Harmer, P. Kilpatrick, and R. Perrott. Managing grid computations: an ORC-based approach. In B. Di Martino, J. Dongarra, and L. T. Yang, editors, *Proc. of the The Fourth International Symposium on Parallel and Distributed Processing and Applications (ISPA'2006)*, 2006. to appear.

[65] The Sun Grid Engine project. http://gridengine.sunsource.net.

[66] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[67] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structures Versus Automata, Proc. of International Workshop*, volume 1043 of Lecture Notes in Computer Science, pages 238–266. Springer-Verlag, 1996.

6 Glossary

Acronym	Meaning
ADL	Architectural Description Language
AM	Application Manager
CAM	Component Application Manager
CCA	Common Component Architecture
CCM	CORBA Component Model
CTL	Computational Tree Logic
DAG	Direct Acyclic Graph
D.PM.xx	Deliverable No. xx of the Programming Model Institute
DoW	Document of Work
FCM	Fractal Component Model
GAM	Grid Abstract Machine
GCM	Grid Component Model
GRAM	Globus Resource Allocation Manager
HOC	Higher Order Components
HOC-SA	Higher Order Component - Service Architecture
INRIA	Institut National de Recherche en Informatique et en Automatique
JPA	Joint Program of Activities
LCC	Life-Cycle Controller
MAM	Module Application Manager
MDS	Monitoring and Discovery System
MPI	Message Passing Interface
NGG	Next Generation Grid (expert group)
NoE	Network of Excellence
OOP	Object Oriented Programming
OS	Operating System
RPC	Remote Procedure Call
SMP	Symmetric Multi Processor
SOAP	Single Object Access Protocol
SPMD	Single Program Multiple Data
QoS	Quality of Service
QUB	Queen's University of Belfast
UDELFT	University of Delft
UNIPI	University of Pisa
UoW	University of Westminster
UPC	Universitad Politecnica de Cataluna
VP	Virtual Processor(s)
WPx	Work Package x
WS	Web Service(s)
WSDL	Web Service Definition Language
WSRF	Web Service Resource Framework
WWU Muenster	University of Muenster