Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST Implementation

M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi, and C. Zoccolo
Deptartment of Computer Science, University of Pisa
Largo Bruno Pontecorvo 3, I-56127, Pisa, Italy
Email: {aldinuc, coppola, bertolli, campa, vannesch, veraldi, zoccolo}@di.unipi.it

***Abstract*— We present the concept of autonomic super-component as a building block for Grid-aware applications. Super-components are parametric, higher-order components exhibiting a well-known parallel behaviour. The proposal of a super-component feature is part of the experience we gained in the implementation of the ASSIST environment, which allows the development of self-configuring and optimising component-based applications following a structured and hierarchical approach. We discuss how such approach to Grid programming influenced the design of the Grid Component Model (GCM).**

I. Introduction

Grids computing platforms offer the option to run complex and multidisciplinary applications, exploiting aggregate software and hardware resources that are physically available at no single computation site. On the other hand, the Grid is a highly dynamic platform, where resources availability changes over time while the program is executing. This makes adaptivity an essential feature in order to achieve high performance and efficiently exploit the available resources [1], [2].

An adaptive application can be re-configured at run-time to tackle the variation of availability and performance of Grid platforms over time, while preserving the semantics of the ongoing computation [3]. Application configuration changes may aim to target different goals, such as enhance robustness, ensure a given level of QoS and security of an application running on the Grid. To pursue these goals in a very dynamic running environment, as Grids are, application adaptation should be automatically triggered by changes of environment status. This scenario finds a natural description in terms of the *autonomic computing* paradigm [4], [5], which indeed has been widely recognised as prerequisite for current and future Grid-aware applications [6], [7], [8]. Also, the autonomic behaviour is a key feature of the forthcoming Grid Component Model (GCM) specification [9]. The GCM is a proposal for a component model oriented to Grid platforms, being developed within the framework of the CoreGRID Network of Excellence (NoE). Part of our contribution to the GCM component model is in the set of abstractions needed to express autonomic behaviour taking into account the aforementioned aspects in a common and consistent way. Our contribution is based on the experiences we have made in the development of the ASSIST parallel programming environment [10], [11] and of the component model developed in the *Grid.it* research project. In this paper we will introduce the notion of *super-component*, i.e. a higher-order parametric component. In ASSIST an autonomic super-component is the result of a hierarchy of structured autonomic components provided with an embedded parallel behaviour. By instancing a super-component, the programmer selects its functionality *and* its parallel behaviour, which is chosen among a number of well-know patterns. This enables the automatic management on-functional aspects of inner components. Last but not least, since a super-component exposes also autonomic features, it is able to manage itself in order to follow self-configuration, self-optimisation, self-healing, and self-protecting targets.

ASSIST shares with GCM many features w.r.t. autonomic behaviour. On the one hand, GCM is currently a proposal for a framework fully supporting autonomic components, in order to realize component-based autonomic Grid applications. Grid.it components, on the other hand, already provide self-optimising and self-configuring behaviour through a hierarchy of user-configurable manager modules, an approach that already enables building HPC Grid applications.

In Sec. II we discuss previous work related to component models and self-adaptive behaviour in Grid programming also related to super-component abstractions. In Sec. III we briefly recap basics of autonomic computing systems. In Sec. IV we describe the architecture of the ASSIST programming environment and the Grid.it component model. We also describes the role of component managers and the application management hierarchy. Section V focuses on the notion of autonomic super-components (self-configuring and self-optimising), which constitutes the main contribution of the paper. In Sec. VI we show experimental results of the self-management features of ASSIST programs. In Sec. VII we relate the ASSIST approach to the ongoing definition of the GCM component model and we highlight their common aspects. Section VIII concludes our presentation and outlines future work directions.

II. Related Work

High-level programming environments for grid aim at moving most of the grid specific efforts needed while developing high-performance grid applications from programmers to programming tools and run time systems. A seminal proposal is

represented by the CORBA Component Model (CCM) [12], followed by the Condor [13] experience from which we were initially inspired in the design of the ASSIST component [10]. GridCCM [14] is an extension of CCM supporting parallel components with distributed data and communication optimisations differing from our approach because of our focus on adaptivity issues. In this sense and with respect to dynamic reconfiguration and re-optimisation, we share common goals with the GrADS project, besides our programming model exhibits a higher lever of abstraction and transparency of adaptation aspects [1], [15].

The ProActive [16] implementation of the Fractal component model [17] proposes a high-level programming toolkit for the Grid supplying program adaptivity. However, adaptivity policies should be explicitly programmed by directly exploiting ProActive adaptation mechanism. Dynaco (Dynamic Adaptation for Components) is another Fractal-based framework that helps in designing and implementing dynamically adaptable components [18].

We also mention Ibis [19] as another Java based programming environment offering a kind of Divide&Conquer super-component notion but not exploiting autonomicity issues. A similar approach is followed by the Higher-Order Components (HOCs) [20], which are components offering the notion of components parameterised with data and code but do not support autonomicity. Thus, a component expresses a behavioural schema that can be instantiated on the target architecture at hand by providing the corresponding code units. As seen below, the idea of having kind of higher-order components is also exploited in the Grid.it component model but we propose a higher level of abstraction where hierarchy takes an important role.

III. Toward and autonomic Grid Component Model

As shown in Fig. 1, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls them. The managed element could be a hardware resource (storage, CPU, etc.), or a software resource, such as a Web service, or a software *component*. Control loops, which are known in optimisation theory since (at least) the mid of the last century, can be used to apply component self-managing. They split the optimisation process into 1) a *monitoring* phase, where the symptoms are collected; 2) an *analysis* phase, where the current status is checked against the goal status; 3) a *planning* phase, where a plan is created to enact the desired alterations according to some policies; and 4) the *execution* phase, which provides the mechanisms to schedule and perform the necessary changes to the system [7], [4]. Truly autonomic systems are years away, although in the nearer term, autonomic functionality will appear in software, especially in very complex systems as Grid-aware applications. In particular, early autonomic system may treat self-optimisation, self-healing, self-configuration and self-protection as distinct aspects, with different solutions that address each one separately.

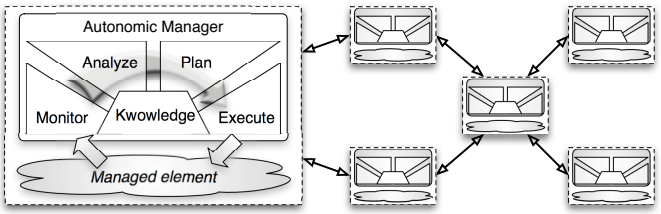

Fig. 1. Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers [4].

As we will see in the following, ASSIST components and super-components may be equipped with a *manager*, while GCM hierarchical components can have *component controllers*. In both models, these managing entities can exploit sub-component's monitoring information (introspection) and autonomic capabilities, at the high and low abstraction levels, in order to achieve global QoS, by assigning contracts as goals to autonomically pursue, as well as to steer the adaptation mechanism of a component when non-local strategies have to be employed to manage less autonomic (legacy) components.

In the following sections we will describe how such aspects are modelled in the ASSIST environment by exploiting *adaptive super-components* as the result of a hierarchy of structured *managers* and how this contribution will be mapped onto the definition of the GCM model.

IV. The ASSIST framework

ASSIST currently supports the Grid.it component model (developed within the Grid.it Italian national project) that shares several features with the forthcoming GCM. A complete porting of ASSIST to meet the GCM is planned in the near future. In the rest of the section we sketch the ASSIST programming environment, starting from its parallel coordination language, its modular architecture and the support for components, to introduce adaptive super-components driven by an ASSIST hierarchy of managers and end up with the description of ASSIST self-managed QoS for performance and self-configuration.

A. The ASSIST Coordination Language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data.

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that run as *Virtual Processes* (VPs) activated by items arriving from the input streams. VPs may synchronise implicitly by activation, or through explicit barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran). VPs virtualise computing resources by de-coupling the definition of parallel/concurrent activities from their instances and their deployment at the implementation level (processes, threads and their mapping onto physical resources).

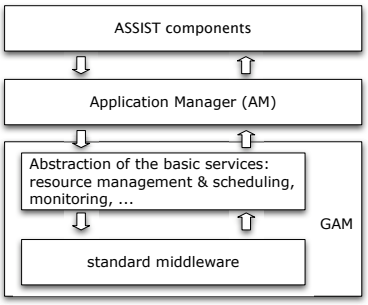

Fig. 2. ASSIST software architecture.

A *parmod* may behave in a data-parallel (e.g. SPMD/forall/apply-to-all) or task-parallel way (e.g. task farm), and it may exploit a distributed shared state, which survives to VPs lifespan. More details on the ASSIST coordination language can be found in [10], [11].

B. The Grid.it Component Model

A single parmod or an arbitrary ASSIST graph of modules can be declared as Grid.it component. A Grid.it component is characterised by *provide* and *use* ports, as well as by *non-functional ports*, which are related to component QoS control. Each port of an ASSIST component may be configured to behave as endpoint of one-way stream connection, RPC method, or event channel. A Grid.it component may also interoperate with Corba/CCM components (via IIOP based RPC) or Web services (via HTTP/SOAP).

The software architecture of the ASSIST component-based parallel programming environment is organised as shown in Fig. 2. The run-time environment of ASSIST 1.3 is implemented on top of a *Grid Abstract Machine* (GAM). The GAM implements *abstract services*, i.e. the functionalities needed by the programming environment to support high-performance, component-based Grid-aware applications. These regard resource discovery, management and monitoring; components deployment, run, and wiring; routing of communications through networks with private addresses. Whether possible, these services rely on the underlying Grid middleware (e.g. Globus), which are just abstracted out at the GAM level. In other cases, GAM services *extend* Grid middleware services (e.g. monitoring) [21].

Grid.it components are characterised by *non-functional* interfaces, which enable introspection and run-time configuration control of the components. These interfaces publicly expose either an RPC or an event-based behaviour, involving subscriptions and following gather of required information. 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), in a correct manner w.r.t. the semantics of the parallel computation. Finally, users may like to stop a component running on a certain grid site, in order to execute an identical copy of it elsewhere, where performance/cost rate may be better. Gracefully stopping a component is a low-level mechanism to implement stateless migration [2], [8].

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

The user or a software manager may leverage on these low-level interfaces to monitor and control the component run-time behaviour. In the latter case, a component with its manager constitute an *autonomic component*. These components hide low-level non-functional interfaces described above, and substitute them with high-level interfaces for:

- the submission of a QoS contract;
- the subscription for QoS contract violations.

The concept of QoS contract is described in Sec. V-A.

C. Component Managers and Management Hierarchy

Both ASSIST modules and Grid.it components can be equipped with managers aiming to dynamically control their behaviour. These managers can be automatically generated by the ASSIST programming environment. Each of these managers behaves as an autonomic manager (see Sec. III). 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.

As an example, Fig. 3 shows an application in which we recognise four components, component a consisting of modules M_1, component b consisting of modules M_2, and component c consisting of modules M_3 and M_4. The whole application exposes a provided port. Each module and component has an associated management entity, respectively a CAM or a MAM, arranged as a tree having the AM as its root. All of these manager should be considered as *logical* entities, nothing prevents each of them to be composed of a set of distributed entities cooperating to achieve a common goal [8].

Each CAM applies control strategies at the level of the associated component, leveraging on non-functional ports of the nested components. Whenever nested components do not exploit any significant mechanism for adaptation and reconfiguration, the CAM can possibly implement strategies based on dynamic component creation and wiring functionalities

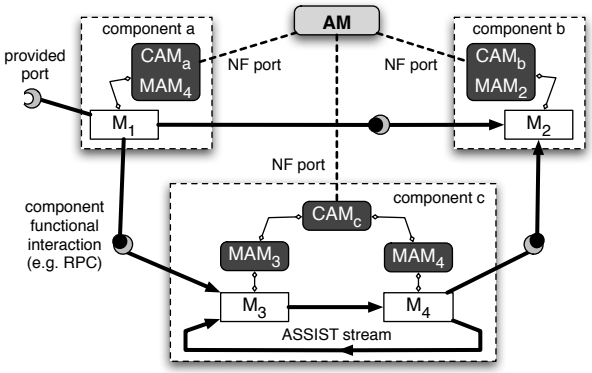

Fig. 3. Four interacting Grid.it components.

provided by the component model. As a concrete example, a component wrapping a legacy MPI program will likely miss adaptation functionalities (its non-functional ports will be no-ops). If the legacy component state can be saved or disregarded, an outer CAM may create on the fly a new instance of the component, with a different configuration and mapping, and substitute the new version to the old one. A CAM can also 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 father CAMs, and can send reconfiguration proposals *(execute)*. The root manager (AM) is the eventual responsible for the final decisions in the global reconfiguration control which, as seen, is a sort of parallel and asynchronous Divide&Conquer strategy applied along the hierarchical management structure.

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. Each component should be equipped with compositional QoS model in order to control if the requested local behaviour propagates bottom-up in the hierarchy in the globally requested behaviour. They should continually adapt their configuration to changes in Grid resources with the goal of sustaining the requested QoS. Notice however that understanding the mapping from local behaviour to global behaviour is a necessary but insufficient condition for controlling autonomic systems. We must also exploit the inverse relationship. The strategy to re-convey a component in the requested behaviour (*plan* and *execute* phases) should be propagated in a top-down fashion in the hierarchy. This means each strategy should be designed in such a way it can be split in different parts, which cooperate to achieve a global goal. These parts are not known in advance since the structure of the hierarchy may change during the run. The definition of a sound set of interaction rules that, once embedded in managers, will induce the desired global behaviour is under investigation. The concept of super-component is a step ahead in this process. A super-component includes a parametric number of components which interact one with each other following a fixed and known pattern. This considerably eases the understanding and the definition of the global to local mapping of component behaviour.

V. Grid.it Super-components

The advantages offered by a hierarchical structure of a component application based on the managers interactions, suggests a further abstraction step leading to the notion of super-component, i.e. a container that can host both other components or super-components. Super-components may be considered as higher-order, parametric components which can be instantiated with other components. They describe common computation paradigms (skeletons, actually [22]).

We have to point out that such common computation paradigms could be also described by manually composing Grid.it components into a graph and by caring about the encoding of the related managers. While the ASSIST compiler can automatically generate templates for the manager of components wired in any way, it cannot yet produce compositional performance models and distributed re-configuration heuristics for any graph of components. The programmer should complete the description of these manager with suitable policies, and this is a complex and error prone programming phase. Due to their well-know parallel structure, super-components eases the compiler task in producing suitable performance models and heuristics, and enable it to produce full-fledged working managers with any programmer intervention, who is not required to care about internal behaviour (such as scheduling and data distribution). This distinguishes Grid.it super-components from other incarnations of higher-order components (e.g. HOCs [20]).

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

a) DAG: it enables the wiring of components/super-components as nodes of a Direct Acyclic Graph, as a generalisation of the a pipeline parallel pattern.

b) Farm: it enables the replication of a given host component/super-component, and is functionally equivalent to the replicated component, exposing the same provided/used functional ports of the host. The farm can of course expose different non-functional ports. Each data item (call/pure data) on the farm provided functional ports is routed to the provided ports of one of the internal replicas. Data items from the replica's used ports are routed on the super-component ones. The replication degree can be changed at run-time via the farm non-functional ports, causing replicas to be spawn/deactivated, exploiting mechanisms and policies analogous to those used in parallel modules, described in this section and in Sec. VI.

Super-components can be used with both event/one-way and RPC-style ports. However, they are particularly suited to be connected via one-way streams, describing a flow of data that is computed along different logical phases. In this case, wiring among components may be done via buffered ports (implemented via distributed queues), enabling multi-site deployment without a strict co-allocation mechanism.

Super-components turn the Grid.it component model into a hierarchical component model, according to GCM specification.

A. QoS Contracts for Autonomic Components

A Grid.it autonomic component accepts (statically or dynamically) a QoS contract via its non-functional interfaces. 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 a contract may be subject to constraints on the amount and on the kind of computing resources.

A Grid.it QoS contract carries a component QoS goal and the description on how it should be achieved. In particular:

- *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 processes resource needs, such as required hardware (platform kind, memory and disk size, network configuration, etc.), required software (O.S., libraries, local services, etc.), and other all strictly required constraints to enforce code correctness.
- *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.

Among all possible QoS goals, Grid.it managers currently support the performance related ones that are achievable through adaptation within each parallel module. Aspects regarding modules coordination, as well as other QoS measures such as reliability, availability, and security are currently under investigation.

The performance models used in the ASSIST framework range from very simple and approximate analytical models, such as the one used to manage task farm parmods, to more complex models derived using advanced mathematical techniques, such as those derived in [2], [23].

As an example, the following is the QoS contract of the experiment in Fig. 4. For more details about Grid.it QoS contracts we refer back to [2], [8].

Perf. features	QL_i (input queue level), QL_o (input queue level), T_{ISM} (ISM service time), T_{OSM} (OSM service time), N_w (number of VPMs), $T_w[i]$ (VPM$_i$ avg. service time), T_p (parmod avg. service time)
Perf. model	$T_p = \max\{T_{ISM}, \sum_{i=1}^{n} T_w[i]/n, T_{OSM}\}$
Goal	$T_p < K$
Deployment	arch = (i686-pc-linux-gnu $\vee$ powerpc-apple-darwin*)
Adapt. policy	goal_based

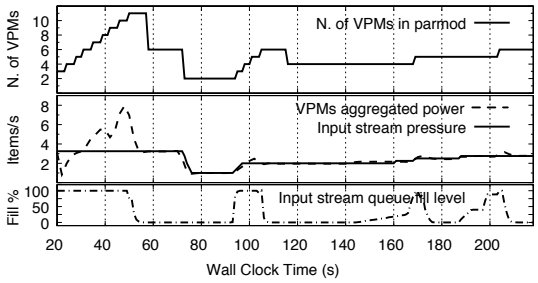

Fig. 4. Experiments on ASSIST adaptivity: Farm reconfigurations guided by QoS contract changes

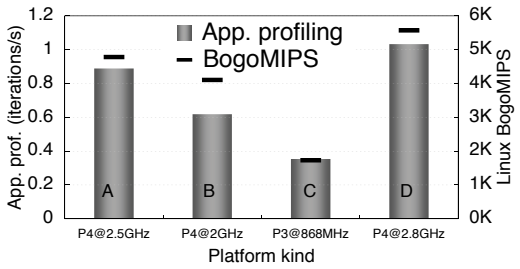

Fig. 5. Experiments on ASSIST adaptivity: Performance of chosen machines on a non-dedicated execution environment

Since ASSIST super-components are pre-defined hierarchical structures of components, the autonomic management discussed can be applied also to such structures, thus allowing the ASSIST framework to expose autonomic super-components.

VI. ASSIST Implementation Results

In this section we will focus on adaptation for ASSIST parallel modules with reference to the satisfaction of user-provided QoS constraints. We will also show quantitative results.

A. ASSIST parmod Adaptivity

As reported in [2], the ASSIST support natively provides for a wide range of dynamic adaptations for parallel modules: new processes can be added/removed at runtime within a parmod, and can be migrated across heterogeneous platforms, using a specifically optimized checkpointing strategy. This allows to exploit remapping strategies to balance the workload in data-parallel computations

No matter when a change request is issued by user or runtime support, the involved module is actually able to reconfigure itself only in special conditions. In such time windows, which are called *reconf-safe* points, the parmod state is consistent, i.e. it is completely defined by the value of its attributes, and no communications involving data are pending. Notably, the runtime does not introduce any additional synchronisation, apart from those required by program semantics. It rather delays reconfiguration execution until the next natural reconf-safe point is reached.

Reconfiguration actions are implemented and optimized for each parmod taking into account its parallel computation

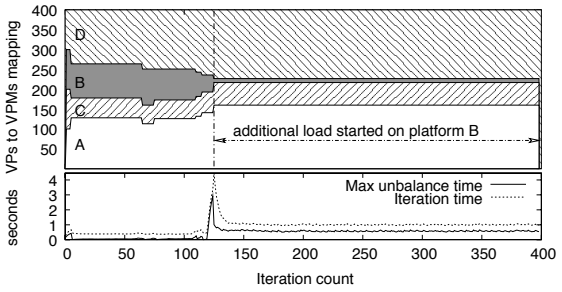

Fig. 6. Experiments on ASSIST adaptivity: Data-parallel rebalancing

semantics. Migration overhead is especially dependent on the knowledge we can infer from the high-level specification of the computation that a parmod provides.

B. Quantitative Results

To evaluate the effectiveness of our approach we report experiences about a farm and a data-parallel parmod (Fig. 4–5). The former farms out a dummy sequential function with 2s average service time; the latter computes an iterative (forall) reduction on internal shared state.

Farm parmod is executed with an initial, relatively strict QoS constraint over the overall module service time (Fig. 4). The QoS contract for the parmod is changed twice by user. The first time, about 70s after computation start, a more relaxed contract is submitted. The runtime support consequently mandates a reduction of the parallelism degree to free the exceeding resources. The second time, QoS gets tighter, thus fresh resources are recruited and new VPM processes launched on them, in order to meet the user requirements.

Data-parallel computation is distributed on four heterogeneous machines in a non-dedicated execution environment (Fig. 5 shows their relative performance). When a processing element is artificially overloaded (Fig. 6, ray-tracing and code compiling starts on node B) the autonomic runtime redistributes the workload trying to rebalance the computation. The support decides to shrink the partition of shared data assigned to the overloaded processing element for the rest of the execution, distributing the exceeding workload proportionally to the performance of the other processing elements.

These experiments show that the approach is feasible for data-parallel computations, and that good results are obtained for the task-parallel ones, for which we found effective adaptation policies.

C. Component-based Applications

Applications are expressed at the highest level, as simple interconnection of black-box components as well as hierarchical compositions of managers, where users only have to define the correct bindings of declared functional interfaces. The functional behaviour of parallel components may be easily and effortless described, naturally exploiting the ASSIST coordination language. The framework supports stream, event and RPC communication paradigms, and consequently provides for component interconnection at run-time by means of declared interface binding. The experiences achieved with the work on ASSIST have naturally driven us to focus more on to the *stream* nature of logical interconnection. Study about the other kinds of support is currently ongoing.

```
<Binding>
 <Components>
  <Component refId="InputComponent">
     <LaunchRef ref="inputPkg"/>
  </Component>
  <Component refId="DataParallelComp">
     <LaunchRef ref="dataParallelPkg"
                dataAAR="calibration"/>
  </Component>
 </Components>
 <Connectors>
  <Stream refId="Channel">
     <BasicType>long</BasicType>
  </Stream>
 </Connectors>
 <Dependencies>
  <Component refId="InputComponent">
     <Produce>
       <Stream refId="Channel" origName="out"/>
     </Produce>
  </Component>
  <Component refId="DataParallelComp">
     <Consume>
       <Stream refId="Channel" origName="src"/>
     </Consume>
  </Component>
  </Dependencies>
</Binding>
```

Fig. 7. A typical composition in a pipeline application

As an example of component composition, Fig. 7 shows a typical layout of a pipeline application, where two stages are defined and a logical connector is created, of type stream. In the simple formalism used to bind components' interfaces together no knowledge of internal component behaviour, nor implementation, is actually required to produce the final packaging of the application. The configuration file exposed can be used as input for a pipeline Application Manager to be executed on Grid machines. Once run, users can interact with the top-level manager to alter the parallel behaviour of the stages or monitor its performance in real-time.

VII. The GCM Approach

The Grid Component Model (GCM in short) has been recently introduced by the Institute of Programming Model of the CoreGRID NoE (WP3), as a unified software component model for Computational Grids, currently at the level of a proposal.

GCM is based on the Fractal component model [17], which is hierarchical. Components can be contained within other components, and the support of a component (its *membrane* in the Fractal terminology) can also be made up of other components. *Component controllers* are primitive Fractal components devoted to controlling their containing component, by means of internal non-functional interfaces which allow them e.g. to alter their host component configuration, and to intercept the communication flow between the inside and the outside of the host component.

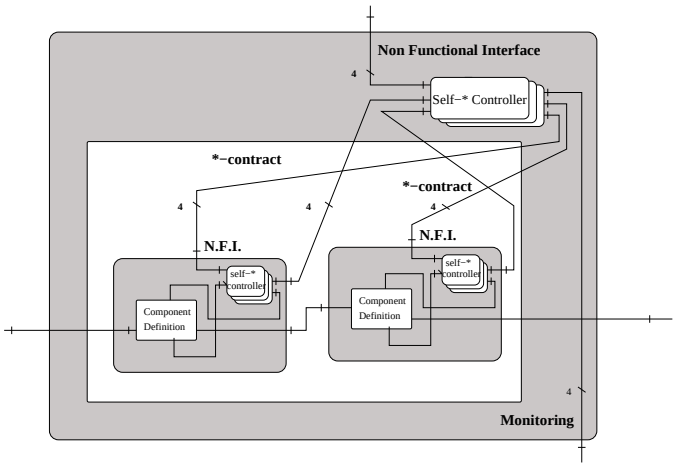

Fig. 8. The hierarchy of controllers inside a GCM composite component.

To enhance component interoperability with legacy code, Fractal considers different levels of compliance of actual components with the implementation framework. GCM applies a similar approach also to autonomic behaviour, with the highest level of compliance being that of fully autonomic components, which implement the whole set of autonomic interfaces. At lower levels of compliance, components can implement a subset of the interfaces needed for autonomic behaviour (e.g. steering or introspection interfaces).

The autonomicity of a GCM component can be characterised w.r.t. two main points:

1) non-functional ports implement the interface through which the user can express, at a high-level, the desired behaviours/properties of the component. Non-functional ports are implemented by a hierarchy of linked *component controllers*, whose structure reflects the component hierarchical structure (see Fig. 8).
2) Each controller is related to a specific aspect of the autonomicity of the component. Controllers are implemented as sub-component of the whole component itself. In order to allow flexible dynamic replacing of support code within a component, *Dynamic* component controllers have been proposed which can be stopped and replaced at run-time, encapsulating specific policies or low-level functionality. The specific implementation of controllers is thus configurable, and defines the autonomic behaviour of the component.

A. Autonomic Component Controllers

The GCM proposal defines a set of separate controllers that hierarchically implement the run-time support of the different autonomicity aspects, exposing interfaces through which contracts can be specified. The organisation of controllers is depicted in Fig. 8:

- The composite component provides a set of non-functional interfaces bound to the controllers of the component itself.
- The controllers of the composite are bound to the controllers of the sub-components by means of subcomponent non-functional interfaces.
- Each sub-component controller directly monitors the subcomponent itself, and, when requested, provides this information to the composite component controller.
- Each sub-component controller directly manages the subcomponent.

As it can be seen, there is a clear similarity between GCM controllers and ASSIST managers, as well as between their hierarchical organisation. However, GCM requires a separate controller for each autonomicity aspect, while in ASSIST more attention is paid on the interactions between managers that coordinate them-selves to reach a common autonomicity goal.

Another common aspect regards the role of non-functional interfaces. Non-functional ports in the GCM proposal are each one related to a specific autonomicity aspect. Examples of the kind of requests acceptable by those interfaces are:

- a new level of performance of the component constraining the service time under a specified threshold, made explicit by a new performance contract (*self-optimisation*)
- a requirement on the fault tolerance level provided, or on the fault recovery mechanism. That contract can require a 99.9% probability of fail-free execution or mandate that a failure should cost less that one hour of computation (*self-healing*).
- a desired level of security for a protocol (e.g. the hardness of a cryptographic code) or a specific constraint on the kind of protocols/resources used (*self-protection*).
- the interface can accept the description of a goal, a parameter, or a procedure to adopt in order to configure the component (*self-configuring*).

The GCM framework proposal does not define how the different contracts can interact, even though it is clear that they do in the general case: for instance, a performance contract can be satisfied if resources are available (thus we may need to self-configure new ones) and as long as the overhead due to fault tolerance is not too high.

It is thus a matter of research to understand how contracts specified at the top of the hierarchy are translated into goals for the leaf components, and how to prevent or control interactions between different controllers, especially if we take into account that dynamic component controller can be replaced at any time. This is a quite powerful feature but it also adds to the complexity of the problem.

As previously mentioned, GCM also supports deriving full autonomic behaviour from non-autonomic components. Controllers can choose to expose outside of the component the steering interfaces that are usually available only from inside, e.g. allowing to modify the parallelism degree of a subcomponent. Devolving control to an outer entity can break autonomicity, but also allows us to develop an overall autonomic composition out of non-autonomic components (having a very limited controller support) by adding external *manager* components which play the autonomic controller role. This

approach is just the same that has been pursued in the design of ASSIST and the Grid.it component model.

VIII. Conclusions and Future Work

In this paper we have presented a general approach to autonomic grid components, which is common to the GCM and the Grid.it component models. We have described the ASSIST architecture with its managing hierarchy ensuring specific QoS aspects and we have introduced the concept of *autonomic super-component* as higher-order parametric component embedding a given parallel behaviour.

From the test results it is clear that ASSIST partially implements the features required in the GCM model, allowing structured design and deployment of component based applications over grids, with high performance and ensuring autonomic control w.r.t. performance and optimisation. There are however differences and open issues which still have to be investigated. GCM models autonomicity with a separate controller for each aspect: it is not yet clear how the different hierarchies related to the aspects will interact.

Even in the simpler design adopted in Grid.it, where managers are not separate for the different aspects, it a critical issue to develop interaction schemes and techniques to break autonomic goals into cooperating and non-conflicting subgoals to be applied to lower levels of a component hierarchy.

Another issue is that even if the ASSIST approach to autonomic performance management is general, we still are more geared toward stream asynchronous communication. While this communication paradigm is quite efficient to exploit over grids, more general models and techniques have to be applied in order to tackle more general application structures.

At the moment we are working to enlarge the set of super-components, and experimenting with different policies and coordination protocols for the manager hierarchy. At the same time, since our experiences will contribute to the analysis and design of the GCM component model, we have planned to enhance the compatibility between the ASSIST/Grid.it environment and GCM, implementing a larger subset of it within our programming environment.

Acknowledgments

This work has been partially supported by Italian national FIRB project no. RBNE01KNFP Grid.it, by Italian national strategic projects *legge 449/97* No. 02.00470.ST97 and 02.00640.ST97, and byt the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

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

[2] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo, "Dynamic reconfiguration of grid-aware applications in ASSIST," in *Proc. of 11th Intl. Euro-Par 2005: Parallel and Distributed Computing*, ser. LNCS, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Springer Verlag, Aug. 2005.

[3] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo, "Parallel program/component adaptivity management," in *Proc. of Intl. PARCO 2005: Parallel Computing*, Sept. 2005.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[5] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, "An architectural approach to autonomic computing," in *Proceedings of the International Conference on Autonomic Computing*, May 2004, pp. 2–9.

[6] *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, Next Generation GRIDs Expert Group, Jan. 2006. [Online]. Available: ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final.pdf

[7] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt, "On adaptability in grid systems," in *Future Generation Grids*, ser. CoreGRID series, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer-Verlag, Nov. 2005.

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

[9] *Deliverable D.PM.02 – Proposals for a Grid Component Model*, CoreGRID NoE deliverable series, Institute on Programming Model, Nov. 2005. [Online]. Available: http://www.coregrid.net

[10] M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, Dec. 2002.

[11] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "ASSIST as a research framework for high-performance grid programming environments," in *Grid Computing: Software environments and Tools*, J. C. Cunha and O. F. Rana, Eds. Springer Verlag, Jan. 2006, ch. 10, pp. 230–256.

[12] The CORBA & CCM home page, http://ditec.um.es/∼dsevilla/ccm/.

[13] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.

[14] A. Denis, C. Pérez, T. Priol, and A. Ribes, "Bringing high performance to the corba component model," in *SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 2004.

[15] S. Vadhiyar and J. Dongarra, "Self adaptability in grid computing," *Concurrency & Computation: Practice & Experience*, vol. 17, no. 2–4, pp. 235–257, 2005.

[16] F. Baude, D. Caromel, and M. Morel, "On hierarchical, parallel and distributed components for Grid programming," in *Workshop on component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds., ICS '04, Saint-Malo, France, June 2005.

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

[18] J. Buisson, F. André, and J.-L. Pazat, "Performance and practicability of dynamic adaptation for parallel computing: An experience feedback from Dynaco," IRISA PI-1782, Rennes, France, Tech. Rep., Feb. 2006.

[19] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a flexible and efficient java-based grid programming environment," *Concurrency & Computation: Practice & Experience*, vol. 17, pp. 1079–1107, 2005.

[20] S. Gorlatch and J. Dünnweber, "From grid middleware to grid applications: Bridging the gap with HOCs," in *Future Generation Grids*, ser. CoreGRID series, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer-Verlag, Nov. 2005.

[21] 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*, ser. CoreGRID series, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer Verlag, Nov. 2005, pp. 217–239.

[22] M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[23] C. Zoccolo, "High-performance component-based programming for heterogeneous computing," Ph.D. dissertation, Dept. Computer Science, Univ. of Pisa, 2005.