16th Euromicro Conference on Parallel, Distributed and Network-Based Processing

Behavioural skeletons in GCM: autonomic management of grid components

Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi
Dept. Computer Science
University of Pisa, Italy
{aldinuc, campa, marcod, vannesch}@di.unipi.it

Peter Kilpatrick
Dept. Computer Science
Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk

Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto
ISTI – CNR
Pisa, Italy
{nicola.tonellotto, domenico.laforenza, patrizio.dazzi}@isti.cnr.it

Abstract

Autonomic management can be used to improve the QoS provided by parallel/distributed applications. We discuss behavioural skeletons introduced in earlier work: rather than relying on programmer ability to design "from scratch" efficient autonomic policies, we encapsulate general autonomic controller features into algorithmic skeletons. Then we leave to the programmer the duty of specifying the parameters needed to specialise the skeletons to the needs of the particular application at hand. This results in the programmer having the ability to fast prototype and tune distributed/parallel applications with non-trivial autonomic management capabilities. We discuss how behavioural skeletons have been implemented in the framework of GCM (the Grid Component Model developed within the CoreGRID NoE and currently being implemented within the GridCOMP STREP project). We present results evaluating the overhead introduced by autonomic management activities as well as the overall behaviour of the skeletons. We also present results achieved with a long running application subject to autonomic management and dynamically adapting to changing features of the target architecture. Overall the results demonstrate both the feasibility of implementing autonomic control via behavioural skeletons and the effectiveness of our sample behavioural skeletons in managing the "functional replication" pattern(s).

1. Introduction

Typical grid architectures are subject to dynamic changes that impact their behaviour [17]. As a consequence, grid applications need to dynamically adapt to the features of the underlying architecture in order to be efficient and/or high performance [3].

In recent years, several research initiatives exploiting component technology [11] have investigated the area of component adaptation, i.e. the process of changing the component for use in different contexts. This process can be either static or dynamic. The basic use of static adaptation covers straightforward but popular methodologies, such as *copy-paste*, *wrapping*, and *OO inheritance*. A more advanced usage covers the case in which adaptation happens at run-time, but here all possible adaptation cases must have been specified at compile time. These systems require that all possible adaptations must be known a priori and must be coded into the application [8, 6]. A second class of systems enables dynamically defined adaptation by allowing adaptations, in the form of code, scripts or rules, to be added, removed or modified at run-time. These systems typically rely on a clear separation of concerns between adaptation and application logic. This approach has recently gained increased impetus in the grid community, especially via its formalisation in terms of the *Autonomic Computing* (AC) paradigm [20, 7, 4]. The AC term is emblematic of a vast *hierarchy* of self-governing systems, many of which consist of many interacting, self-governing components that in turn comprise a number of interacting, self-governing components at the next level down [18]. An autonomic component will typically consist of one or more managed components coupled with a single autonomic manager that controls them. To pursue its goal, the manager may trigger an adaptation of the managed components to react to a run-time change of application QoS requirements or to the platform status.

In this regard, an assembly of self-managed components

[0]This research is carried out under the FP6 Network of Excellence CoreGRID and the FP6 GridCOMP project funded by the European Commission (Contract IST-2002-004265 and FP6-034442).

0-7695-3089-3/08 $25.00 © 2008 IEEE
DOI 10.1109/PDP.2008.46

implements, via their managers, a distributed algorithm that manages the entire application. Several existing programming frameworks aim to ease this task by providing a set of mechanisms to dynamically install reactive rules within autonomic managers. These rules are typically specified as a collection of `when-`*event*`-if-`*cond*`-then-`*act* clauses, where *event* is raised by the monitoring of component internal or external activity (e.g. the component server interface received a request, and the platform running a component exceeded a threshold load, respectively); *cond* is an expression over component internal attributes (e.g. component life-cycle status); *act* represents an adaptation action (e.g. create, destroy a component, wire, unwire components, notify events to another component's manager) [12, 15, 19].

In this work, we briefly introduce *behavioural skeletons* [2], we discuss implementation issues of behavioural skeletons, and finally we present some experimental results. Behavioural skeletons represent an innovative way to describe autonomic components in the GCM framework. They aim to model recurring patterns of component assemblies that can be (either statically or dynamically) equipped with correct and effective management strategies with respect to a given management goal. Behavioural skeletons help the application designer to *a)* design component assemblies that can be effectively reused, and *b)* cope with management complexity by providing a component with an explicit context with respect to top-down design (i.e. component nesting).

In the rest of the paper we briefly discuss some related work (Section 2), we introduce GCM (Section 3) and we briefly outline autonomic management in GCM (Section 4). Then, we discuss implementation issues of behavioural skeletons in the framework of GCM (Section 5). Finally, we discuss experimental results achieved in the framework of the GridCOMP project (Section 6).

2. Related work

The idea of autonomic management of parallel/distributed/grid applications is present in several programming frameworks, although in different flavours. ASSIST [26, 3], AutoMate [23], K-Components [14], SAFRAN [13] and finally the forthcoming CoreGRID Component Model (GCM) [11] all include autonomic management features. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component model [21]. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids, and all except ASSIST are component based. While the current work extends the GCM model with the skeleton concept, it could equally have built upon K-Components or the AutoMate framework as all provide distributed system based component frameworks with autonomic capability.

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

3. GCM: the Grid Component Model

GCM is a hierarchical component model explicitly designed to support component-based autonomic applications in highly dynamic and heterogeneous distributed platforms, such as grids. It is currently under development by the partners of the EU CoreGRID Network of Excellence[1]. A companion EU STREP project, GridCOMP [2] is currently developing an open source implementation of GCM and preliminary versions are already available for download as embedded modules in the ProActive middleware suite[3]. GCM builds on the Fractal component model [21] and exhibits three prominent features: hierarchical composition, collective interactions and autonomic management. The full specification of GCM can be found in [11].

Hierarchical composition A GCM component is composed of two main parts: the *membrane* and the *content*. The membrane is an abstract entity that embodies the control behaviour associated with a component, including the mediation of incoming and outgoing invocations of content entities. The content may include either the code directly implementing functional component behaviour (*primitive*) or other components (*composite*). In the latter case, we refer to the included components as the *inner components*. GCM components, as Fractal ones, can be hierarchically nested to any level. Component nesting represents the *implemented_by* relationship. Composite components are first class citizens in GCM and, once designed and implemented, they cannot be distinguished from primitive, non-composite ones.

Collective interactions GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RPC-like" use/provide ports (or

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

client/server interfaces), GCM allows data, stream and event ports to be used in component interaction. Both static and dynamic wiring between dual interfaces is supported. Each interface may expose several *operations* of different types. Furthermore, collective interaction patterns (communication mechanisms) are also supported. In particular, composite components may benefit from customisable one-to-many and many-to-one functional interfaces to distribute requests arriving to one component's port to many inner components and gather requests from many inner components to a single outgoing port.

Autonomic management Autonomic management aims to attack the complexity which entangles the management of complex systems (as Grid applications are) by equipping their parts with self-management facilities [18]. GCM is therefore assumed to provide several levels of autonomic managers in components, that take care of the non-functional features of the component programs. GCM components thus have two kinds of interfaces: functional and non-functional ones. The functional interfaces host all those ports concerned with implementation of the functional features of the component. The non-functional interfaces host all those ports needed to support the component management activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation. Each GCM component therefore contains an *Autonomic Manager* (AM), interacting with other managers in other components via the component non-functional interfaces. The AM implements the autonomic cycle via a simple program based on the reactive rules described above. In this, the AM leverages on component controllers for the *event* monitoring and the execution of reconfiguration *actions*. In GCM, the latter controller is called the *Autonomic Behaviour Controller* (ABC). This controller exposes server-only non-functional interfaces, which can be accessed either from the AM or an external component that logically surrogates the AM strategy. We call *passive* a GCM component exhibiting just the ABC, whereas we call *active* a GCM component exhibiting both the ABC and the AM.

Beside these features, that clearly differentiate GCM from other notable component models such as CCA [9] and CCM [22], GCM also uses an *ADL* to support component deployment, supports *interoperability* with state of the art environments such as Web Services, and provides a set of *compliance* levels that range more or less from POJOs (Plain Old Java Objects) to autonomic composite components.

4. Adaptation in GCM

In GCM, autonomic behaviour of components is implemented through AMs (the Autonomic Managers) and ABCs (Autonomic Behaviour Controllers). Programmers may write their own AM and ABC implementation using the mechanisms provided by the GCM run time. This is similar to what programmers do when using other AC paradigms, such as the ones mentioned in Sec. 2. This requires substantial knowledge on the part of programmers, relating to both autonomic control principles and to the component model itself. Without such detailed knowledge it is very difficult to develop efficient and effective autonomic controllers/managers. We recognise, however, that common patterns of autonomic management can be adopted in grid applications, and, to this end, we have introduced *behavioural skeletons* [2].

4.1 Behavioural skeletons

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

They represent a specialisation of the algorithmic skeleton concept for component management [10]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. Behavioural skeletons, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit skeletons' inherent semantics to design sound self-management schemes of parallel components.

As a byproduct, behavioural skeletons allow categorisation of GCM designers and programmers into three classes. They are, in increasing degree of expertise and decreasing cardinality:

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

2. *GCM expert users*: they use behavioural skeletons overriding the AM management strategy. However, the specialisation does not involve the ABC and thus does not require specific knowledge about the GCM membrane implementation.

3. *GCM skeleton designers*: they introduce new behavioural skeletons or classes of them. To this end, the design and development of a brand new ABC might be

required. This may involve the definition of new interfaces for the ABC, the implementation of the ABC itself, together with its wiring with other controllers, and the design and wiring of new interceptors. Obviously, this requires quite a deep knowledge of the particular GCM implementation.

Due to the hierarchical nature of GCM, behavioural skeletons can be identified with a composite component with no loss of generality (identifying skeletons as particular higher-order components [16]). Since component composition is defined independently from behavioural skeletons, they do not represent the exclusive means of expressing applications, but can be freely mixed with non-skeletal components. In this setting, a behavioural skeleton is a composite component that *a)* exposes a description of its functional behaviour, *b)* establishes a parametric orchestration schema of inner components, *c)* may carry constraints that inner components are required to comply with, and *d)* may encompass a number of pre-defined plans to cope with a given self-management goal.

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

In both cases two features of behavioural skeletons are exploited: on the one hand, the skeletons exhibit an explicit higher-order functional semantics that delimits the skeleton usage and definition domain. On the other hand the skeletons describe parametric interaction patterns and can be designed in such a way that parameters affect non-functional behaviour but are invariant for functional behaviour.

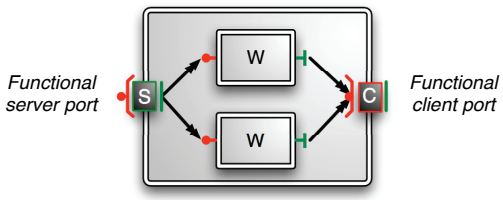

Figure 1. Functional replication behavioural skeleton schema

In [2] we introduced a simple set of behavioural skeletons, mainly modelling *functional replication* parallel patterns. We assumed our skeletons have two functional interfaces: a one-to-many stream server S, and a many-to-one client stream interface C. They accept requests on the server interface and then dispatch the (partial) requests to a number of instances of an inner component W which may propagate results outside the skeleton via C interface (see Figure 1). We assume that replicas of W can safely forget the internal state between different calls. For example, the component has just a transient internal state and/or stores persistent data via an external data-base component.

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

In this case, in our farm behavioural skeleton, a stream of tasks is absorbed by a *unicast* S. Then each task is computed by one instance of W and the result is sent to C, which collects results according to a *from-any* policy. This skeleton can be equipped with a self-optimising policy as the number of Ws can be dynamically changed in a sound way since they are stateless. The typical QoS goal is to keep a given limit (possibly dynamically changing) of served requests in a time frame. Therefore, the AM just checks the average time tasks need to traverse the skeleton, and possibly reacts by creating/destroying instances of W, and wiring/unwiring them to/from the interfaces.

Once available, the task farm behavioural skeleton can be conveniently and easily adapted to cover other common patterns of parallel computation. For example, data parallel computations can be captured by simply modifying the behaviour associated with the S and C interfaces. In a data parallel computation a stream of tasks is absorbed by a *scatter* S. Each of the tasks appearing is split into (possibly overlapping) partitions, which are distributed to replicas of W to be computed. The results computed by the W are *gathered* and assembled by C in a single item, which is eventually delivered onto the output stream. As in the previous case, the number of Ws can be dynamically changed (between different requests) in a sound way since they are stateless. In addition to the previous case, the skeleton can be equipped with a self-configuration goal, e.g. resource balancing and tuning (e.g. disk space, load, memory usage), that can be achieved by changing the partition-worker mapping in S (and C, accordingly).

The task farm (and data parallel) behavioural skeletons just outlined can be easily modified to the case in which the S is an RPC interface. In this case, the C interface can be either an RPC interface or missing. Also, the stateless functional replication idea can be extended to the stateful case

by requiring inner components Ws to expose suitable methods to serialise, read and write the internal state. A suitable manipulation of the serialised state enables the reconfiguration of workers (also in the data-parallel scenario [3]).

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

5. Autonomic Components: design and implementation

The two main characteristics of autonomic components are the ability to self-manage and to cooperate with other autonomic components to achieve a common goal, such as guaranteeing a given behaviour of an entire component-based application. In the light of this, viewing the management of a single component as an atomic feature enables design of its management (to a certain extent) in isolation. The management of a single component is therefore considered a *logically centralised* activity. Components will be able to interact with other components according to well-defined protocols described by management *interaction patterns*, which are established by the component model.

5.1. The management of a GCM component

The management of a single component is characterised by its ability to make non-trivial decisions. Thus GCM components are differentiated as being *passive* or *active*, with the following meanings:

Passive A component exposes non-functional operations enabling introspection (state and sensors) and dynamic reconfiguration. These operations exhibit a parametric but deterministic behaviour. The operation semantics is not underpinned by a decision making process (i.e. does not implement any optimisation strategy), but can only be constrained by specific pre-conditions that, when not satisfied, may nullify an operation request. All components should implement at least a reflection mechanism that may be queried about the list and the type of exposed operations.

Active A component exhibits self-managing behaviour, that is a further set of autonomic capabilities built on top of passive level functionality. The process incarnates the autonomic management process: monitor, analyse, plan, execute. The *monitoring* phase is supported by introspective operations, while the *executing* phase is supported by re-configuring operations described above.

In the architecture of GCM components, these two features are implemented within the Autonomic Behaviour Controller (ABC) and Autonomic Manager (AM), respectively. Since the management is a logically centralised activity, a single copy of each of them can appear in a component. Notice that, this does not prevent a parallel implementation of them for different reasons, such as fault-tolerance or performance. A passive component implements just the ABC, whereas an active component implements both the ABC and the AM. The following relationship holds

$$Comp <: PassiveComp <: ActiveComp$$

where $<:$ is a subtyping relation. This is described in the GCM specification by increasing values of conformance levels [11].

GCM Passive Autonomic Components The ABC and the AM represent two successive levels of abstraction of component management. As mentioned above, the ABC implements operations for component reconfiguration and monitoring. The design of these operations is strictly related to membrane structure and implementation, and therefore the choice of implementing the ABC as a controller in the membrane was the more obvious and natural. Within the membrane, the ABC can access all the services exposed by sub-component controllers, such as that related to life cycle and binding, in order to implement correct reconfiguration protocols. In general, these protocols depend on component structure and behaviour. However, in the case of behavioural skeletons they depend almost solely on the skeleton family and not on the particular skeleton. In this regard, the ABC effectively abstracts out management operations for behavioural skeletons.

For the sake of exemplification we use the functional replication family. In this case, the reconfiguration operations require the addition/removal of workers as well as the tuning of distribution/collection strategies used to distribute and collect tasks and results to and from the workers. The worker addition/removal operations can be used to change the parallelism degree of the component as well to remap workers on different processing elements and/or platforms. The distribution/collection tuning operations can be used to throttle and balance the resource usage of workers, such as CPU, memory and IO. The introspection operations involve querying component status with respect to one or more predefined QoS metrics. The component status is generally ob-

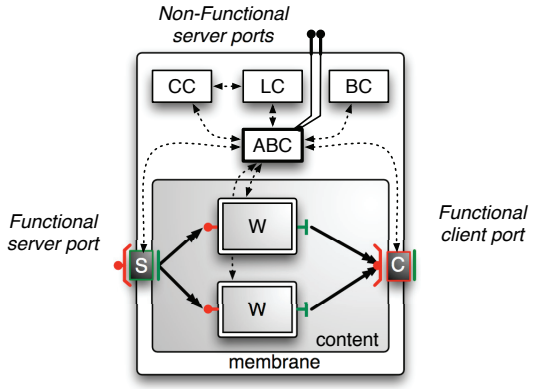

Figure 2. GCM: membrane and content (CC is the content controller, LC the lifecycle controller and BC is the binding controller).

tained as a harmonised measure involving component status and inner component status.

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

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

1. *Stop.* The ABC requires the *Lifecycle Controller* (LC) to stop all the components. To this end, the LC retrieves from the *Content Controller* (CC) the list of inner components $\mathsf{W}_1 \cdots \mathsf{W}_n$, and then issues a `stop` on them.

2. *Type Inspection.* All the $\mathsf{W}_1 \cdots \mathsf{W}_n$ have the same type. The ABC retrieves from the CC the list of inner components $\mathsf{W}_1 \cdots \mathsf{W}_n$, then retrieves TypeOf($\mathsf{W}_1$).

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

4. *Bind.* The component server interface S is wired to newly created $\mathsf{W}_{n+1} \cdots \mathsf{W}_{n+k}$ inner components via the *Binding Controller* (BC). $\mathsf{W}_{n+1} \cdots \mathsf{W}_{n+k}$, in turn, wire their client interfaces to the component collective client interface C. The process requires the inspection of the types of the interfaces of W_1 that is used again as a template for all W_i.

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

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

`get_measure(m)` *Semantics:* Query the component about the current status of the measure m, which may depend on the status of the inner components (possibly involving other measures) and the membrane status.
Examples: Transactions per unit time, load balancing, number of up-and-running workers, etc.

1. *Collect Workers' Measures.* The ABC retrieves from the CC the list of inner components $\mathsf{W}_1 \cdots \mathsf{W}_n$, then issues a `get_measure(m)` on each.

2. *Collect Membrane Measures.* The ABC queries membrane sensors relating to the particular metric m.

3. *Harmonise Measures.* Measures acquired from workers and from the membrane are harmonised by using a m-dependent function (e.g. average, maximum, etc.).

4. *Return.* Return a failure code if some of the previous operations failed (e.g. sensor not implemented in inner components); return monitor information otherwise.

GCM Active Autonomic components The operations implemented in the ABC can be arbitrarily complex; however, they do not involve any decision making process. In general, each of them implements a protocol that is a simple list of actions. On the contrary, the AM is expected to enforce a contractually specified QoS. To this end the AM should decide *if* a reconfiguration is needed, and if so, *which* reconfiguration plan can re-establish contract validity [1]. Furthermore, as we shall see in Sec. 5.2, the AM should also determine if the contract violation is due to the managed component or is the byproduct of other components' malfunction. The architecture of an active GCM component is shown in Fig. 3.

The AM accepts a QoS contract[4], which is currently defined as pair $\langle V, E \rangle$, where V is a set of variables representing the measures the AM can evaluate (via the ABC), and E is a mathematical expression over these variables that might include the $\min$ and $\max$ operator over a finite domain. The set of V determines the minimum set of measures the AM should be able to monitor to accept the contract. The E encodes the constraints and goal the AM is required to pursue. This encoding can be realised in many different ways provided E can be evaluated in finite time and possibly quite efficiently. As an example, we are currently using a simple functional notation with no recursion.

Having accepted a QoS contract, the AM iteratively checks its validity, and in the case that it appears broken, evaluates a number of pre-defined reconfiguration plans. Each reconfiguration plan consists of a sequence of actions

[4] the notion of *QoS contract* is still the subject of further investigations and possible refinements. The one discussed here is the bare minimum necessary to discuss AM behaviour and implementation.

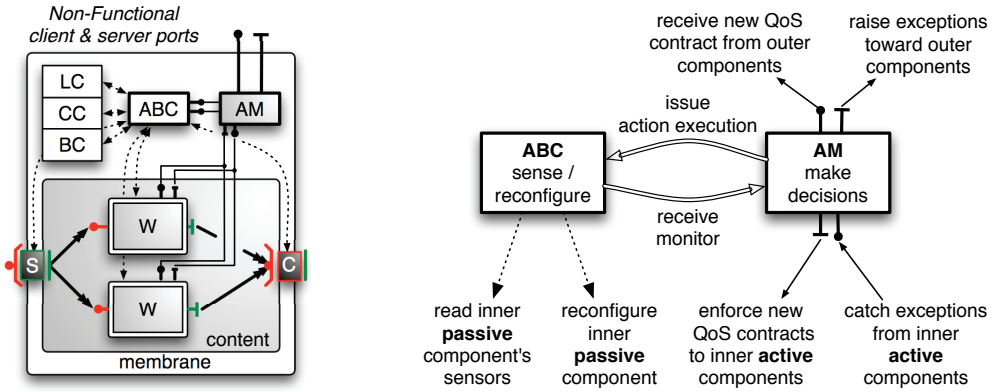

Figure 3. Left) GCM active component architecture. Right) ABC and AM interaction.

(to be executed via the ABC), and a QoS forecast formula. This formula allows the value of a subset of V after the reconfiguration to be forecast. The AM instantiates in turn all reconfiguration plans obtaining, for each plan, a set of forecast values. A plan is marked as *valid* if the set of V updated with forecast values satisfies the QoS contract. Among the valid plans the AM heuristically chooses the reconfiguration plan to be executed. If no reconfiguration plan is valid, an exception is raised.

As is clear, the main difficulty in the AM definition is the specification of a reconfiguration plan. In the general case, the reconfiguration plans, and especially their forecast formulae, are strictly related to the behaviour of a particular component. As discussed in Sec. 4, behavioural skeletons enable the definition of reusable reconfiguration plans by categorising and restricting component behaviour in families and skeletons.

5.2. Cooperative management

The ultimate goal of QoS management is to guarantee user intentions despite software and environmental instabilities and malfunctions. To this end the management of a whole system should be coordinated to achieve a common goal. In general, we envisage a component-based system as a graph, whose nodes are components, and edges are relations among them, such as data dependency, management, geographic locality, etc. Different relations can be kept distinct by a proper labelling of edges. In this work we restrict the focus to two relations which are of particular interest for GCM: *used_by* and the *implemented_by* (see Sec. 4). Since the GCM is a hierarchical model, the nesting relation naturally defines the implemented_by relationship. In particular, the application structure along the nesting relation describes a tree whose nodes represent components (leaves are primitive components) and edges represent their nesting. In this case, the management of a composite component C is cooperatively performed by the AM_C of the component itself and the AM_{C_i} of the child components $C_i, i = 1..n$. In the case where inner components are passive, the cooperation is really one of control by the outer component: services exposed by the ABC_{C_i} are called by the ABC_C.

Conceptually, non-functional properties modelling runtime behaviour of the whole hierarchy can be synthesised in a bottom-up fashion: the behaviour of a composite component depends on the behaviour of its nested components. Management actions and QoS contracts should be projected along the tree in a top-down fashion: the users usually would like to declare a global goal they expect from an application. This matches the idea of submitting a contract at the root of tree. A fully autonomic system should automatically split the global goal into sub-goals that should then be forced on inner components.

On the whole, each GCM component enforces local decisions. When a contract violation is detected, its AM tries autonomously to re-establish the contract to a valid status by re-configuring its membrane or inner components. In the event that it cannot (no valid plan), it raises an event to its father component, thus increasing the extent of the reconfiguration. The overall behaviour enforces the maximum locality of reconfigurations, which is a highly desirable property in a distributed system, since it eases the mapping of components onto the network of platforms, that usually exhibit a hierarchical nature in terms of uniformity of resources and latency/bandwidth of networks (cluster of clusters).

Observe that cooperation between components is unavoidable even in very simplistic applications. Let us consider an example:

Producer-filter-consumer Let us assume the application sketched in Fig. 4 has the final goal to generate, render, and display a video with a given minimum number of frames/sec ($FPS > k$). The contract is split into three identical contracts since the property should be enforced on

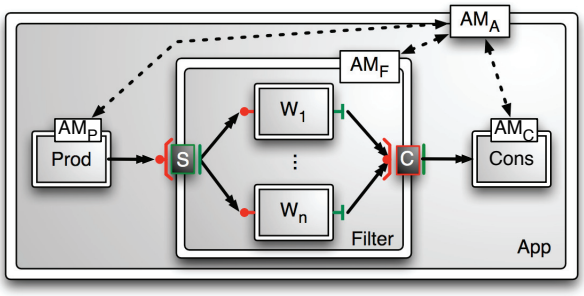

Figure 4. Producer-filter-consumer with parallel filter (farm skeleton).

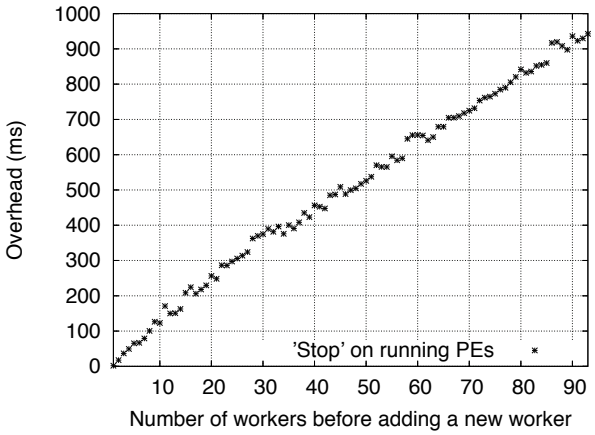

Figure 5. Reconfiguration overhead: Stop.

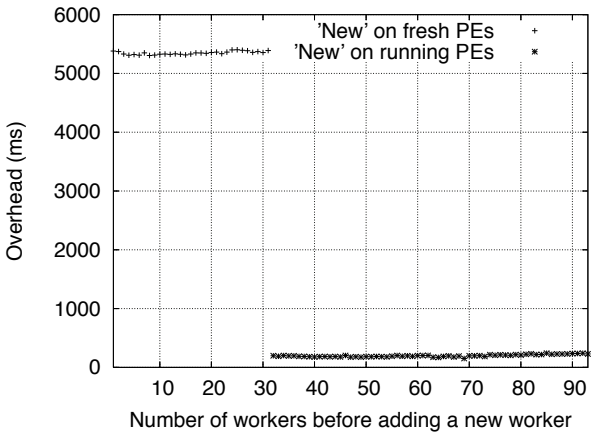

Figure 6. Reconfiguration overhead: New.

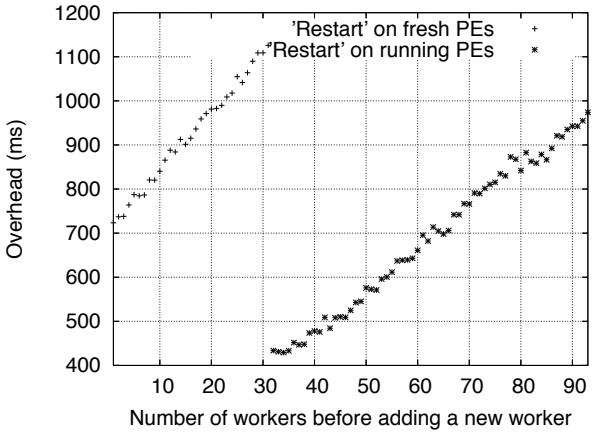

Figure 7. Reconfiguration overhead: Restart.

all stages in order to hold globally. The rendering (filter) has been parallelised since it is the most CPU-demanding stage. Two common problems of such applications are a transient overload of platform where $W_1 \cdots W_n$ are running, or an increased complexity of scene to be rendered. These events may lead to a violation of QoS contract at the AM_F. In this case it may increase the number of workers (mapped on fresh machines) to deal with the insufficient aggregate power of already running resources. In many cases this will *locally* solve the problem. However, a slightly more sophisticated contract should consider also the input and output channels. In particular the filter stage might be not rendering enough frames because it does not receive enough scenes to render. In this case the AM_F can detect the local violation, but cannot locally solve the problem. As a matter of fact, no plan involving a change of parallelism degree can solve this problem. AM_F can just signal the problem to a higher level AM_A, which can try to remap the input channel to a faster link, or simply signal to the end user that the contract is not satisfied.

6. Experiments

Experiments have been conducted on the current prototype of the GCM that is under development in the GridCOMP STREP project [24]. The prototype, which is being developed on top of ProActive middleware [25], includes almost all of the features described in this paper. Experimental data is measured on the application shown in Fig. 4. The experiments mainly aim to assess the overhead due to management and reconfiguration. For the sake of reproducibility, the experiments have been run on a cluster instead of a more heterogeneous grid. The cluster includes 31 nodes (1 Intel P3@800MHz core per node) wired with a fast Ethernet. Workers are allocated in the cluster in a round robin fashion with up to 3 workers per node (for a total of 93 workers). Note however, the experimental code can run on

any distributed platform supported by the ProActive middleware.

Figures 5, 6, and 7 respectively show the time spent on the farm behavioural skeleton (filter) for the *stop*, *new* and *restart* ABC services described in Sec. 5.1. This time is application overhead, since none of the workers can accept new tasks during the process. In the figures, a point k in the X-axis describes the overhead due to *stop/new/restart* in the adaptation of the running program from a k to $k+1$ worker configuration. As highlighted by the curves in Fig. 5 and 7 the overhead of *stop* and *restart* is linear with respect to the number of workers involved in the operations. This is mainly due to a linear time barrier within the Life cycle Controller (LC), which is an inherent part of the underlying ProActive middleware. Note that adaptation process does not strictly require such a barrier. Both stopping all the workers and linear time synchronisation are peculiarities of the current GCM implementation on top of the ProActive middleware, and not of the farm behavioural skeleton, which can be implemented avoiding both problems. In addition, the creation of a new worker can be executed outside the critical path by using a speculative creation. These techniques have been used in the ASSIST implementation of the farm skeleton, which exhibits a constant overhead [5].

Figure 6 shows the time spent for the *new* ABC operation (see Sec. 5.1). Again, in this case, the time is overhead. The experiment measures the creation of a single worker, and thus the times measured are almost independent of the number of workers pre-existing the new one.

As highlighted by the Fig. 6 and 7 the overhead of the *new* and *restart* operations is much higher in the case where a fresh platform is involved (number of workers less than 32). The difference is mainly due to the additional time for Java remote class loading.

The results of the last experiment are presented in Fig. 8. It describes the behaviour of the application over quite a long run that includes several self-triggered reconfigurations. The application is provided with a QoS contract that enforces the production of a minimum of 1.5 results per second (tasks/s). During the run, an increasing number of platforms are externally overloaded with an artificial load (C++ compilation). The top half of the figure reports the measured average throughput of the filter stage, and the QoS contract. The bottom half of the figure reports the number of overloaded machines along the run, and the corresponding increase of workers of the filter stage. Initially the throughput of the filter stage is abundantly higher than requested (~ 3.5 tasks/s); but it decreases when more machines are overloaded. As soon as the contract is violated, the AM reacts by adding more workers.

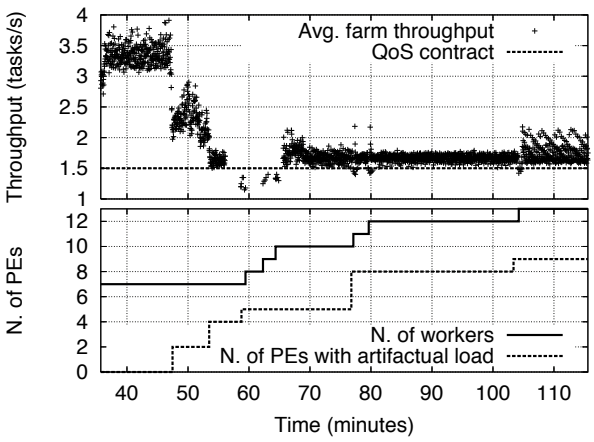

Figure 8. Self-optimisation experiment.

7. Conclusions

We discussed how behavioural skeletons, already introduced in a previous work, can be implemented in the framework of the GCM component model. Behavioural skeletons provide the programmer with the ability to implement autonomic managers completely taking care of the parallelism exploitation details by simply instantiating existing skeletons and by providing suitable, functional parameters.

In particular, in this work we analysed several issues related to the implementation of a functional replication behavioural skeleton. We presented experimental results that demonstrate both the typical overheads involved in autonomic management operations and also dynamic adaptation occurring during execution of a long-running application. Finally, we discussed the results achieved when running an application exploiting instances of our behavioural skeletons and we showed how the skeletons used may take decisions at the appropriate time to maintain the application behaviour within the limits stated by the user with a specific performance contract. The whole experiments have been performed using GCM components and behavioural skeletons, as being designed and implemented in the framework of the CoreGRID and GridCOMP projects.

To our knowledge, no other similar results available yet. As the behavioural skeleton approach has been proven feasible and effective, we are currently working to provide further skeletons, to refine the implementation and to perform more experiments involving the use case applications identified in the framework of the GridCOMP project, that include data intensive as well as transaction processing applications.

References

[1] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 89–96, Germany, Dec. 2005. John von Neumann Institute for Computing.

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

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

[4] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST grid-aware components. In B. D. Martino and S. Venticinque, editors, *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pages 221–230, Montbéliard, France, Feb. 2006. IEEE.

[5] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. 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 Processing*, volume 3648 of *LNCS*, pages 771–781. Springer, Aug. 2005.

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

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

[8] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Cosell. A declarative approach for designing and developing adaptive components. In *Proc. of the 15th Intl. Conference on Automated Software Engineering*, pages 111–119. IEEE, 2000.

[9] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *HPDC*, pages 51–, 2000.

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

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

[12] P.-C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In J.-B. Stefani, I. Demeure, and D. Hagimont, editors, *Proc. of the Intl. Conf. on Distributed Applications and Interoperable Systems*, number 2893 in LNCS, pages 1–14, Paris, France, 2003. Springer.

[13] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In W. Löwe and M. Südholt, editors, *Proc. of the 5th Intl Symposium Software on Composition (SC 2006)*, volume 4089 of *LNCS*, pages 82–97, Vienna, Austria, Mar. 2006. Springer.

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

[15] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. A platform supporting coordinated adaptation in mobile systems. In *Proc. of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*, pages 128–137, Callicoon, New York, U.S., June 2002. IEEE.

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

[17] 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.

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

[19] S. Neema, T. Bapty, J. Gray, and A. S. Gokhale. Generators for synthesis of QoS adaptation in distributed real-time embedded systems. In *Proc. of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 236–251. Springer, 2002.

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

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

[22] OMG. Corba components. Technical Report Document formal/02-06-65, OMG, June 2002.

[23] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling autonomic applications on the Grid. *Cluster Computing*, 9(2):161–174, 2006.

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

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

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