Towards hierarchical management of autonomic components: a case study

Marco Aldinucci, Marco Danelutto
Computer Science Department
University of Pisa
Pisa, Italy
Email: {aldinuc, marcod}@di.unipi.it

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

Abstract

We address the issue of autonomic management in hierarchical component-based distributed systems. The long term aim is to provide a modelling framework for autonomic management in which QoS goals can be defined, plans for system adaptation described and proofs of achievement of goals by (sequences of) adaptations furnished. Here we present an early step on this path. We restrict our focus to skeleton-based systems in order to exploit their well-defined structure. The autonomic cycle is described using the Orc system orchestration language while the plans are presented as structural modifications together with associated costs and benefits. A case study is presented to illustrate the interaction of managers to maintain QoS goals for throughput under varying conditions of resource availability.

1. Introduction

Increasingly it is becoming clear that the handling of non-functional concerns, such as fault-tolerance, performance, security, etc., in parallel and distributed systems presents some of the most challenging issues in their development. The fact that such systems are often long-lived and require on-going management of such concerns exacerbates the problem. Recognition that the scale of many such systems precludes user interaction as a basis for this management has turned the focus to autonomic systems [1].

The idea of autonomic management of distributed applications is present in several programming frameworks in various flavours. ASSIST [2], AutoMate [3], SAFRAN [4], and GCM [5] all include autonomic management features. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids. While these systems considerably ease the production of autonomic applications, the task of developing the management code remains onerous in the general case, and the inclusion of management code tends to act as a hindrance to component re-use.

To address the difficulty of developing such autonomic managers for distributed systems while retaining a degree of re-use capability, we previously introduced the behavioural skeleton concept, a composite skeleton-based component that exposes a description of its functional behaviour and establishes a parametric orchestration schema of its inner components as well [6]. The idea was to marry the algorithm skeleton idea with that of autonomic management in such a way that the structure derived from the use of skeletons would ease the burden of management and, to a degree, facilitate re-use. (To a certain extent the ASSIST framework – the forerunner of the behavioural skeleton – was premised on the same combination, but there the separation of management code was not so pronounced.) In that earlier work our aim was to establish the feasibility of the approach and the focus of [6] was on implementation issues.

In this work we aim to address that complexity of autonomic management by building an abstract framework which can provide a vehicle for investigating the nature of interactions among managers and the establishment of manager goals. In particular we will look at *hierarchical* autonomic management. In [7] the authors argue that, while a hierarchy is not the only possible arrangement of managers in large systems, in many ways it is the most natural, following as it does the often chosen top-down approach to system development and also the organization of the personnel in many IT departments.

Here we define in a formal way the goals of hierarchical autonomic management and the policies used to implement it. The former are the *contracts*, either provided by the end user or automatically derived by the autonomic manager(s) within the application; the latter comprise the general algorithm used to combine the activities of several managers, each belonging to a separate *component* (behavioural skeleton) of the application, in such a way that a global target can be effectively pursued.

We first present a general overview of hierarchical autonomic management (Sec. 2); then we introduce the farm- and pipe-based skeleton system that we use as a basis for our hierarchical framework (Sec. 3); in section 4 we provide an Orc model of the autonomic managers involved, which can form the basis for reasoning about the overall system behavior; and finally we discuss some examples to illustrate the use of adaptation plans for modifying skeleton systems where service time of the overall application is the measure to be autonomically controlled (Sec. 5).

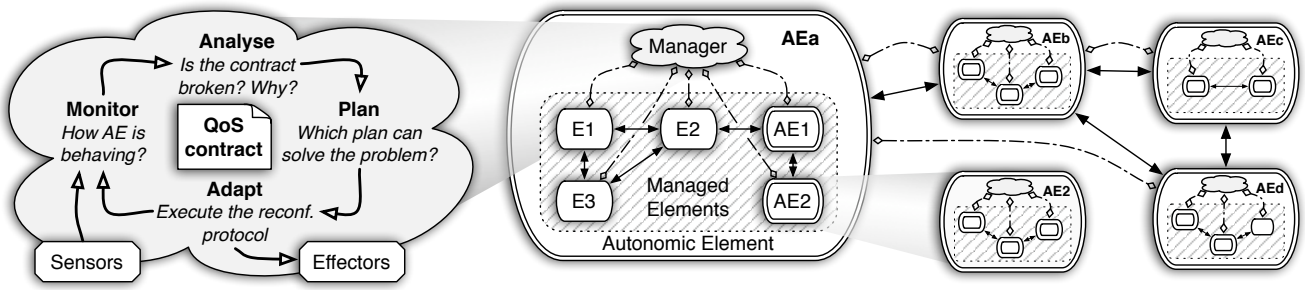

Figure 1. An autonomic system at several levels of details. Clouds represents managers; boxes with double border are Autonomic Elements (AEx); boxes with single border are non-autonomic Elements (Ex); solid arrows represent data exchanges; dashed arrows represent management overlay.

2. Autonomic management and contracts

Following the introduction of its original Autonomic Computing manifesto [8], IBM in 2003 refined the *Autonomic Computing* term to be representative of *"a hierarchy of self-governing systems, which may consist of many interacting, self-governing elements that in turn comprise a number of interacting, self-governing elements at the next level down"* [1]. The term derives from the body's autonomic nervous system which controls key functions without conscious awareness.

In general, autonomic management aims to attack the complexity which entangles the management of complex systems (as distributed and Grid applications are) by equipping their parts with self-management facilities. Autonomic computing tries to tackle the problem with the often-quoted four "selves": self-configuration, self-healing, self-optimisation, self-protection (and, as a combination of all, self-management). Truly autonomic systems are years away, although autonomic functionality is appearing more and more frequently in the design of complex systems such as grid applications and cloud frameworks. These systems currently treat the four "selves" as distinct aspects, with different solutions addressing each one separately, with their integration to achieve multi-purpose autonomic management still an open problem [9].

As sketched in Fig. 1, an autonomic element typically consists of one or more managed elements controlled by a single autonomic manager. Those managed elements might be either standard elements (i.e. hardware or software components) or other autonomic elements. To pursue its *local* 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. Also, the manager may interact with other similar managers to pursue a *global* goal.

An autonomic manager contains a control loop that implements four functions: monitor, analyse, plan, and adapt[1]. The monitor function collects details about the resources being managed. The analyse function takes the collected information and determines where changes are required. The plan function is responsible for generating any required plans, and the adapt function takes necessary actions to implement planned changes [1]. Each manager pursues a goal specified in a *QoS contract* [2]. In the context of autonomic component models, such as GCM [5], it is convenient to regard each component as having its own manager. We consider three kinds of manager, with increasing degree of autonomic capability:

1) The *empty* manager, which exhibits no ports and thus cannot even be monitored.
2) The *passive* manager, which may *provide* ports to outer components and may *use* ports of the inner components: these ports implement monitoring functionality.
3) The *active* manager, which may *provide* to and *use* ports of both outer and inner components; these ports may implement either monitoring activity or the installation of a new QoS contract, which may induce the execution of a reconfiguration operation.

We categorize components in the same way; we also assume that component nesting may be in a non-increasing order of management capability (outer smarter than inner). As a result, autonomic managers can be arranged in a hierarchy where the management policies can be unfolded, by way of QoS contracts, along component nesting. The topmost contract reflects the user intention; a contract within the hierarchy can be derived from its parent. This covers the cases in which each component controls either directly or indirectly its inner components. In the former case, as sketched in Fig. 2, the management is strictly hierarchic and contracts can be unfolded along a tree, whereas in the

1. we use "adapt" rather than the standard "execute" to avoid confusion with execution of core functionality.

latter case the contracts can be unfolded along a hierarchical graph. In this work, we assume the first case since it is simpler and it still may provide a level of efficiency in distributed systems since the levels of the tree can be easily associated with different levels of component coupling and geographic location.

2.1. Manager blueprint

We consider the active manager since the passive manager can be obtained by reduction of its functionality. The manager (component) collects monitor data from managed elements (inner components); collection may happen as a polling process (via manager *use* ports) or event-based notification (via manager *provide* ports). The manager M_C of the component C collects from its n controlled components a set of monitor values $\overline{m}_i(\circlearrowleft_t)$ that are assigned to variables m_i at autonomic cycle t, each of them being a list of variables, i.e. $i = 0 \cdots n, \exists j_i \in \mathbb{N} : m_i = [m_i^1, \ldots, m_i^{j_i}]$, where m_0 represents the state of C itself. These values can be either received by way of events or polled from controlled components. Also, C receives a QoS contract from its parent.

The contract carries a *contract predicate* $\mathcal{CP}(m_0, \ldots, m_n)$ expressed within a suitable logic, which is decidable and efficient for the evaluation of predicates representing typical QoS requirements. In general, the identification of such a logic may present a significant challenge; for simplicity, we assume here m_i are lists of variables ranging over $\mathbb{R} \cup \perp$[2], and $\mathcal{CP}$ is a ground formula whose terms express inequalities between variables m_i and constants. $\mathcal{CP}$ may be evaluated to true (valid) or false (broken). In the former case no actions are required. In the latter case, the manager considers its reconfiguration plans. Each plan consists in a sequence of adaptation actions, whose execution leads to an expected change of one or more m_i. The manager in turn simulates the execution of these plans in order to determine if some of them may produce a set of values $m_i(\circlearrowleft_{t+1})$ of the variables m_i that satisfy $\mathcal{CP}$. This task can be heuristically performed according to a goal function $\mathcal{G}(m_0, \ldots, m_n)$ that quantifies the quality of the solution. Once a plan is chosen, it is executed. If no plan is suitable, the manager assumes it cannot autonomously solve the problem and triggers an event toward the manager at the next level up. The event may carry the value of some of monitor variables $m_0, \ldots, m_n$ (or some elaboration of them) to the next level up. A plan is a sequence of actions with one of the following goals:

1) push a new contract into controlled components;
2) reconfigure the assembly of managed components;

2. where $\perp$ represents undefined variables (e.g. due to a timeout in their gathering).

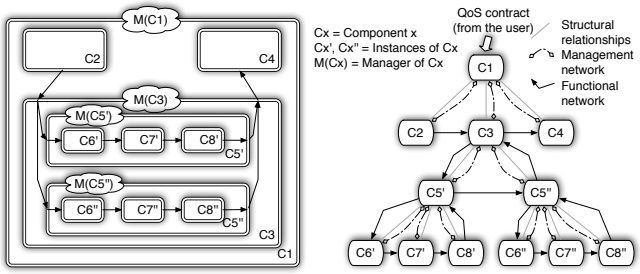

Figure 2. An autonomic component application and its structure. Three relationships between components are highlighted.

3) raise an event toward companion managers (reachable non-lower level managers).

A plan comprises three parts:

- **Actions.** These actions realise non-trivial protocols such as remapping a component into a new platform, creating and deploying a new component instance, sending a message in the proper format (examples can be found in [6]).
- **Expected benefit.** The benefit that each plan is supposed to deliver is described by a set of equations describing the variation of m_i at some future iteration of the autonomic cycle, as an example $m_0(\circlearrowleft_{t+3}) = g(m_1(\circlearrowleft_t), m_2(\circlearrowleft_t)), m_1(\circlearrowleft_{t+3}) = f(m_2(\circlearrowleft_t))$. These equations should be easily calculable in the logic chosen for the autonomic management.
- **Expected overhead.** Enacting a plan may have an immediate overhead in terms of some m_i (e.g. reconfiguration time and number of resources) that should be forecast in the same way as the expected benefit (but paid just once).

Observe that the list of available operations used in actions, expected benefit and overhead for a given component are in general very dependent upon the features and implementation of the component. Thus the design of *general* plans is likely to be a complex activity. Behavioural skeletons cope with this complexity by narrowing this generality and offering standard plans for families of components that exhibit similar behaviour.

3. Skeleton framework

In the remainder of this paper we discuss the concepts introduced in Sec. 2 in the framework of hierarchical autonomic management of *structured* parallel programs. The programs considered are developed using a simple but significant skeleton system including pipelines and farms. We consider a framework such that the "structure" of the parallel program considered is a term of the grammar sketched in Fig. 3 left (we assume here that "Seq" will be a generic

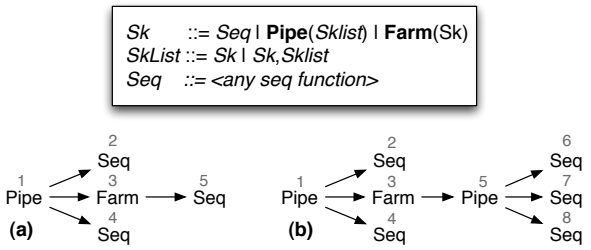

Figure 3. Skeleton grammar (upper part) and sample skeleton programs (lower part; shaded identifiers are included to allow reference to program parts from within the text).

function (procedure or method without side effects) written in some sequential language).

Several considerations contributed to the decision to consider autonomic management of structured programs:

i) A large proportion of distributed/Grid applications can be modelled using this skeleton set.

ii) Several skeleton programming environments support this kind of structured programs (including Muesli [10] and eSkel [11]).

iii) By limiting the kind (structure) of the parallel programs considered we succeeded in designing effective hierarchical autonomic management strategies that exploit program structure. This is fully compliant with the algorithmic skeleton vision: restrict the parallel forms considered to a few that have been demonstrated useful and reusable and capitalise on the narrower solution space deriving from this restriction. Although we consider only two kinds of parallel patterns, we are able to investigate different autonomic management policies (with the associated hierarchical composition problems) due to the fact that pipelines and farms behave differently when propagating performance contracts to their parameter (inner) skeletons.

iv) By considering autonomic management of skeleton-based parallel programs we proceed further in the refinement of the *behavioural skeleton* concept as introduced in [6]. However, the results discussed in that paper could have been derived also for more generic parallel programs, whose interaction graph is not a plain tree, as in the skeleton case, although the derivation would have been a little bit more difficult.

Typical programs of the skeleton framework considered here are those represented in Fig. 3. Program (a) corresponds to the classical schema of a three stage computation where the first stage is sequential and produces a stream of tasks processed by the second, parallel stage, possibly reading some input task file from disk. The third stage sequentially post-processes the computed tasks, possibly storing them to the disk. Program (b) represents a further parallelisation of program (a) if the programmer recognises that the second stage can be further split into three separate stages, thus augmenting the amount of parallelism that the implementation of the skeleton environment can use. It is worth pointing out that program (b) perfectly models the structure of most of the use-case applications that are currently being considered in the framework of the GridCOMP EU STREP project to validate the design and implementation of the GCM component model [12].

4. Autonomic cycle

We present an Orc [13] model of the autonomic management activities as outlined in Sec. 2. In particular we present a high-level view of a behavioural skeleton manager implementing the autonomic cycle. In [6] we argued that Orc is suitable for describing behaviour, as it is a language for orchestrating distributed services. The purpose of the Orc model is twofold [14]: first to describe precisely the activity of the managers, while maintaining readability so the description can act as a design artifact; and second to exploit the precision to allow (semi-)formal reasoning about properties of the model.

Briefly, the Orc constructs used below are these: $\gg$ denotes sequential composition; $> v >$ denotes sequential composition with passing of a value; $\mid$ denotes parallel execution, with obvious extension to *N* workers indexed by *i*; and $let(x)$ `where` $x :\in (P \mid Q)$ denotes the publication (*let*) of *x* where *x* is the *first* value to be returned by $P \mid Q$, and the termination of further execution of $P \mid Q$.

4.1. Behavioural Skeleton Manager

A behavioural skeleton comprises a *skeleton* together with a concurrently executing *manager* that acts to ensure a *contract* is maintained.

$$BSkel(skeleton, manager, contract) \triangleq \\ skeleton \mid manager(skeleton, contract)$$

The skeleton provides the functionality and the manager enacts the autonomic cycle – monitor, analyse, plan and adapt.

$$manager(sk, c) \triangleq \\ distribute(sk, c) > sk1 > monitor(sk1) > m > \\ analyse(sk1, m) > (b, p) > \\ (\ \texttt{if}(b) \gg adapt(sk1, p) > sk2 > manager(sk2, c) \\ \mid \texttt{if}(\neg b) \gg let(contViol) \gg passiveMode > c1 > \\ manager(sk1, c1))$$

The *manager* distributes the contract, c, appropriately over the skeleton, sk, producing a new skeleton, *sk1*, identical

in structure to *sk* but with each sub-component having a suitable contract. During a given cycle, t, the *monitor* gathers the monitor values, m, from the components of the skeleton and passes these to *analyse*. The result of this analysis is a pair (b, p). b is a boolean indicating if an appropriate plan has been identified. If so, the *adapt* stage modifies the skeleton accordingly, producing *sk2* and management continues with this new skeletal structure; if not, then we have a situation where the contract is being violated but no suitable adaptation plan can be identified. In this latter case, the manager can only pass a "contract violation" message up the hierarchy and enter a passive mode in which it no longer enacts the autonomic cycle (it simply acts as a passive manager, responsive to the monitoring of its parent) but awaits a new contract at which point it can return to active management.
(Note: in the case that the contract is not broken then the plan will simply be "carry on as before" and the *adapt* stage will have no effect, so *sk2* = *sk1*.)

The challenge lies in defining the actions of a manager (distribute, monitor, etc.) in a notation suitable for the sort of semi-formal reasoning we espoused in [14]. This may, for example, involve enhancing Orc's simple data types to allow for the description of plans. This is ongoing work. Here, to give a flavour, we consider a farm behavioural skeleton, and in particular the adaptation action required following a breach of contract.

$$BSkel(farm(N), contract) \triangleq$$
$$\quad farm(N) \mid manager(farm(N), contract)$$

A farm consists of a set of workers, W_i, executing in parallel.

$$farm(N) \triangleq \ (\mid 1 \leq i \leq N : W_i)$$

A worker receives a task on an in channel, processes it using $W_i\_execute(x)$ and sends the result on an out channel; it then recurs. Receipt of an input task may be interrupted (by the manager), in which case the worker terminates (described by 0).

$$W_i \triangleq$$
$$(\ \ \texttt{if}(b) \gg (W_i\_execute(x) > y > out.put(y) \gg W_i)$$
$$\mid \texttt{if}(\neg b) \gg 0)$$
$$\quad \texttt{where}\ \ (x, b) :\in$$
$$\quad\quad (\quad in.get > y > let(y, true)$$
$$\quad\quad\ \mid Interrupt_i.get > y > let(y, false))$$

Adaptation is achieved by terminating each of the workers with an interrupt and instantiating a new farm with an additional worker.

$$adapt(farm(N), plan) \triangleq$$
$$(\texttt{if}(plan = addworker) \gg let(y) \gg farm(N+1)$$
$$\quad\quad \texttt{where}\ \ (\forall i :: y_i :\in Interrupt_i.set)$$

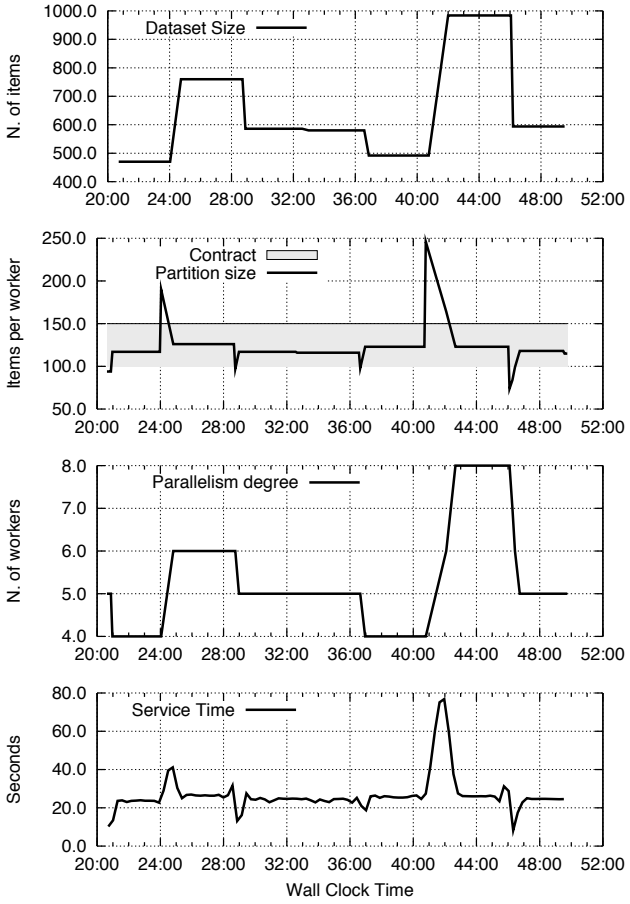

Figure 4. Data parallel behavioural skeleton at work

The synchronization following the publication of all $Interrupt_i.set$ signals is achieved using the barrier synchronization idea of [13].

5. Use cases

We consider a typical use case and discuss how hierarchical autonomic management proceeds for some typical situations. We assume service time (throughput) is the measure to optimise, i.e. we will try to shorten as much as possible the average time spent by the application to deliver two consecutive items on its output. We denote service time for component C_i as T_{C_i}. We also assume that parallelism exploitation in our application is performed using skeletons modelled by appropriate behavioural skeleton components [6].

The results discussed here have been achieved by simulating the manager hierarchy as described in Sec. 2 and modelled through the Orc code of Sec. 4. To simulate more accurately the behaviour of single autonomic managers we actually used data from experiments made using the autonomic managers of a task farm and data parallel behavioural

skeletons developed within the GridCOMP project. Those managers are implemented according to the guidelines given in Section 2. The kind of management implemented is demonstrated in Fig. 4. Here a sequence of variable sized input data sets is presented to an application exploiting parallelism by using a data parallel behavioural skeleton. The application is a biometric identification application matching scanned fingerprints against a (large) fingerprint database in real time [15]. The size of input data, in terms of number of data items, is presented on the first plot of the Figure. The manager is given a contract asking to keep a data set dimension *per worker* constant (the bar in the second graph, plotting the dimension of the single worker data set). The manager reacts to changes in the input data size by adding and removing workers from the executor worker strings (third plot) and as a result, the service time of the behavioural skeleton varies as expected (fourth plot). This kind of experiment allowed us to validate the policies and mechanisms implemented and used within a single manager and to evaluate the reaction times as well.

We consider program (b) of Fig. 3 as the use case. This program uses pipeline and farm behavioural skeletons: therefore we define both the theoretical service time and the measures m_i of interest, according to what is stated in Sec. 2.

The service time of a pipeline with stages $C_1, \ldots, C_k$ is

$$T_{\text{pipeline}(C_1,\ldots,C_k)} = max\{T_{C_1}, \ldots, T_{C_k}\} \qquad (1)$$

as the slowest stage obviously represents the bottleneck for the whole computation. In this case, the different T_{C_i} represent instances of the measures m_i monitored on the inner components (they are obtained by querying inner pipeline components' passive managers). The service time of a farm with n_w workers instantiated from C_w, is given in turn by

$$T_{\text{farm}(C_w,n_w)} = \sum_{j=1..n} T_{C_i}/n_{\text{w}}^2 \qquad (2)$$

as the average service time of the single worker is

$$\sum_{j=1..n} T_{C_i}/n_{\text{w}}$$

and we have n_w workers in the farm.

Also, we define how contracts are used in pipelines and farms. If a pipeline is given a contract $\mathcal{CP}$ (usually of the form $\mathcal{CP} = K_{\text{low}} \leq T_{\text{pipeline}} \leq K_{\text{high}}$) the pipeline manager *propagates* the contract to all the inner components, i.e. asks the managers of the pipeline stages to ensure the very same contract. If a farm is given a contract concerning the service time (usually $\mathcal{CP} = T_{\text{farm}} \simeq IT_{\text{farm}}$, where IT_{farm} is the inter-arrival time for the farm), the contract propagated to the inner components (the worker component instances) is instead an $\text{optimize}(T_C)$. This is due to the fact the service time is modelled as shown in (1) and (2), respectively.

Let us consider what is happening in the sample use case we consider here. First we define the kind (and number) of managers in the program, as well as the contracts and the m_i of interest. Table 1 describes the structure of the program in terms of managers, contracts and monitor values received from inner (nested) skeletons or computed by the manager themselves. Contracts as described in Table 1 derive from a single contract established by the end user: this contract states that the service time of the overall program has to be in the interval $[K_{\text{low}}, K_{\text{high}}]$.

Pipeline skeleton managers pass their contract to inner nodes. Farm skeleton managers, instead, forward to the worker managers the contract simply stating that service time is to be optimised. A farm contract is ensured by the farm manager, possibly by adding (removing) workers once the workers do their best to optimise service time (e.g. pipe managers keep the pipe stage balanced in terms of T_S).

Then, we considered some feasible plans for the managers. We present distinct plans for pipelines and farms, as sketched in Table 2.

Under these hypotheses, we assumed an initial configuration of the parallel program that satisfies all the contracts and we simulated hierarchical autonomic management in the case that a violation of the contract is detected by the manager in C_5 due to a violation of contract by node C_7 (e.g. due to some additional load started on the C_7 resources). We considered two situations: in the former (C.1), computing resources not yet allocated to the program computation are available and some of these new resources are faster than those used to allocate program computations. In the latter case (C.2), further resources are available but not faster than the current ones. The sequence of events in the simulation of the two cases goes as follows.

C.1 The C_5 manager monitors a contract violation on node C_7. This happens in a single instance among the C_5 ones instantiated as workers of the C_3 farm. Considering the Orc manager model presented in Sec. 4, this means in the sequence $\ldots > monitor(sk1) > m > analyse(sk1, m) > (b, p) > \ldots$ the $monitor$ collects a new, sub-contractual, service time from C_7 and $analyse(sk1, m)$ detects that the current contract of the pipeline is broken, as stages are no longer balanced. The C_5 manager therefore looks for plans $(\ldots > (b, p) > adapt(sk1, p) > (sk2, c) > \ldots)$ and identifies a new configuration ($sk2$). In particular, $\mathsf{PL}_{\mathrm{P}_1}$ is considered first. As we assume to have new, faster resources available, the plan can be applied and its application via *adapt* results in a modified structure, $sk2$. The adaptation plan is therefore implemented and the manager begins a new iteration of the autonomic management cycle with the new configuration ($manager(sk2, c)$).

C.2 The initial steps are similar to those performed in the previous case. However, due to the unavailability of

Component	Type	Manager Contract	m_i
C_1	active pipe	$K_{\text{low}} \leq T_{\text{self}} \leq K_{\text{high}}$ (*user defined*)	$K_{\text{low}}, K_{\text{high}}$ constants; $T_{C_2}, T_{C_3}, T_{C_4}$ monitored $T_{\text{self}} = \max\{T_{C_2}, T_{C_3}, T_{C_4}\}$ $[\uparrow]$ $\mathcal{CP}_{C_2} = \mathcal{CP}$; $\mathcal{CP}_{C_3} = \mathcal{CP}$; $\mathcal{CP}_{C_4} = \mathcal{CP}$ $[\downarrow]$
C_3	active farm	$(\mathcal{CP}_{\text{super}}) \wedge (IT_{\text{self}} \leq T_{\text{self}})$ (*derived)*	IT_{self} = request inter-arrival time; n_{self} = #workers let C_j children of $C_3, 1 \leq j \leq n_{\text{self}}$: T_{C_j} monitored $T_{\text{self}} = \sum_{j=1..n_{\text{self}}} T_{C_j}/n^2_{\text{self}}$; $[\uparrow]$ $\mathcal{CP}_{C_j}$ = optimise(T_{C_j}); $[\downarrow]$
C_5	active pipe	$\mathcal{CP}_{\text{super}}$ (*derived*)	$T_{C_6}, T_{C_7}, T_{C_8}$ monitored $T_{\text{self}} = \max\{T_{C_6}, T_{C_7}, T_{C_8}\}$; $[\uparrow]$ $\mathcal{CP}_6$ = null; $\mathcal{CP}_7$ = null; $\mathcal{CP}_8$ = null; $[\downarrow]$
$C_{2,4,6,7,8}$	passive seq	none	provide $T_{C_{2,4,6,7,8}}$ via NF port (respectively)

Table 1. Managers and contracts for program (b) of Fig. 3. *self* denotes the component itself; *super* denotes the father component; $[\uparrow]$ denotes local monitor values synthesised from the monitoring of inner components; $[\downarrow]$ denotes contract predicates that the current level of the hierarchy enforces at the lower levels (inner components).

	Plan	Expected Cost	Expected Benefit
PL_{F_1}	move the slower worker C_w to a faster platform, if any	cost(stop(C_w); deploy(C_w); start(C_w))	decrease service time. $T_{\text{farm}}(\circlearrowleft_{t+h})$ = $\delta T_{C_w}(\circlearrowleft_t)$, $0 \leq \delta \leq 1$ speed difference between the platforms
PL_{F_2}	increase parallelism degree (allocate k new workers)	cost(deploy(C_{w_j}); start(C_{w_j})) for $j = 1..k$ instances	decrease service time. $T_{\text{farm}}(\circlearrowleft_{t+h})$ = $\delta T_{\text{farm}}(\circlearrowleft_t)$ $\delta = n/(n+k)$
PL_{F_3}	decrease parallelism degree (de-allocate k workers)	cost(stop(C_{w_j})) for $j = 1..k$ instances	increase service time. $T_{\text{farm}}(\circlearrowleft_{t+h})$ = $\delta T_{\text{farm}}(\circlearrowleft_t)$ $\delta = (n+k)/n$
PL_{F_4}	raise violation	0 *(negligible)*	none
PL_{P_1}	move stage (C_s) with maximum T to a faster resource, if any	cost(stop(C_s); deploy(C_s); start(C_s))	decrease service time. $T_{\text{pipe}}(\circlearrowleft_{t+h})$ = $\delta T_{\text{pipe}}(\circlearrowleft_t)$, $0 \leq \delta \leq 1$ speed difference between the platforms if $max\{T_{C_S}, T_{\text{pipe}}(C_1, \ldots, C_{s-1}, C_{s+1}, \ldots, C_k)\} = T_{C_S}$, otherwise $\delta = 1$
PL_{P_2}	collapse adjacent stages C_s, C_{s+1}	cost(stop(C_s); deploy(C_s); start(C_s)) for C_s and C_{s+1}	decrease resource usage $n = n - 1$. increase service time. $T_{\text{pipe}}(\circlearrowleft_{t+h})$ = $\delta + T_{\text{pipe}}(\circlearrowleft_t)$, $\delta = 0$ iff $T_{C_s} + T_{C_{s+1}} \leq T_{\text{pipe}}(\circlearrowleft_t)$, $\delta = T_{C_s} + T_{C_{s+1}} - T_{\text{pipe}}(\circlearrowleft_t)$ otherwise
PL_{P_3}	raise violation	0 *(negligible)*	none

Table 2. Initial set of feasible plans considered for pipeline and farm managers.

new faster resources PL_{P_1} cannot be applied. PL_{P_2} cannot be applied either as it cannot optimise service time. At this point the C_5 manager reports a failure to the C_3 manager. The C_3 manager becomes aware of the failure while executing $\ldots > sk1 > monitor(sk1) > m > analyse(sk1, m) > \ldots$. While in the $analyse$ step, a violation of the global C_3 contract can be detected consequent to the violation reported by C_5. Therefore the C_3 manager will consider plan PL_{F_2}, which is eventually implemented ($> (b, p) > adapt(sk1, p) > sk2 >$), the number of workers in the farm is increased and a new autonomic cycle is started (plan PL_{F_1} is not applicable for the same reason we did not succeeded applying PL_{P_1}).

Different behaviour is achieved in the two cases, with the same management schema. In case C.1 contract violation is dealt with locally. In case C.2 the contract violation cannot be dealt with locally so the parent manager is informed and

it will eventually perform appropriate corrective actions to solve the problem. After adaptation no direct verification of the effect of the actions performed is made before the beginning of the next autonomic cycle. It may be the case that the action taken does not solve the problem, due to small (but unavoidable) inaccuracy of the cost/benefit models in the plans or to rapid variations in the target architecture load. Actions have to be considered to avoid thrashing (continuing "oscillating" adaptations). However, in the case of relatively small inaccuracy of the cost/benefit models, adaptation will simply require some additional cycles to be achieved, provided the inaccuracy in the models does not affect the goal of the adaptation process.

6. Conclusions

In this paper, we have outlined a framework suitable for modelling hierarchical autonomic management in the general case of applications build out of interacting autonomic components. We have further specialized the framework to the domain of behavioural skeletons where the inherent skeleton structure eases the burden of propagating contracts within the structure and devising adaptation plans. Orc modelling is used to describe management activity, thus permitting precise formulation and opening the possibility of reasoning about the models to predict behaviour prior to implementation.

We discussed simulation results showing that effective hierarchical management can be implemented when (as an example) service time is to be autonomically optimized. These simulation results have been obtained using single (non-hierarchical) managers controlling GCM behavioural skeletons developed within the CoreGRID and GridCOMP EU FP6 projects. They demonstrate that management decisions can be used to make effective structural modifications in response to contract breaches. The extension of these experiments to hierarchical management as described here is underway and preliminary results are consistent with the theoretical framework presented. This extension will be reported on in a forthcoming paper.

References

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

[2] M. Aldinucci and M. Danelutto, "Algorithmic skeletons meeting grids," *Parallel Computing*, vol. 32, no. 7, pp. 449–462, 2006.

[3] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri, "AutoMate: Enabling autonomic applications on the Grid," *Cluster Computing*, vol. 9, no. 2, pp. 161–174, 2006.

[4] P.-C. David and T. Ledoux, "An aspect-oriented approach for developing self-adaptive fractal components," in *Proc. of the 5th Intl Symposium Software on Composition (SC 2006)*, ser. LNCS, W. Löwe and M. Südholt, Eds., vol. 4089. Vienna, Austria: Springer, Mar. 2006, pp. 82–97.

[5] *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, CoreGRID NoE deliverable series, Institute on Programming Model, Feb. 2007. [Online]. Available: http://www.coregrid.net

[6] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick, "Behavioural skeletons in GCM: autonomic management of grid components," in *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, D. E. Baz, J. Bourgeois, and F. Spies, Eds. Toulouse, France: IEEE, Feb. 2008, pp. 54–63.

[7] P. Brittenham, R. R. Cutlip, C. Draper, B. A. Miller, S. Choudhary, and M. Perazolo, "IT service management architecture and autonomic computing," *IBM Systems Journal*, vol. 46, no. 3, pp. 565–681, 2007.

[8] *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM, 2001. [Online]. Available: http://www.research.ibm.com/autonomic/manifesto/

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

[10] H. Kuchen, "A skeleton library," in *Proc. of 8th Euro-Par 2002 Parallel Processing*, ser. LNCS, B. Monien and R. Feldman, Eds., vol. 2400. Paderborn, Germany: Springer, Aug. 2002, pp. 620–629.

[11] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, "Flexible skeletal programming with eSkel," in *Proc. of 11th Euro-Par 2005 Parallel Processing*, ser. LNCS, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Lisboa, Portugal: Springer, Aug. 2005, pp. 761–770.

[12] GridCOMP Project, "Grid Programming with Components, An Advanced Component Platform for an Effective Invisible Grid," 2008. [Online]. Available: http://gridcomp.ercim.org

[13] J. Misra and W. R. Cook, "Computation orchestration: A basis for wide-area computing," *Software and Systems Modeling*, vol. 6, no. 1, pp. 82–110, Mar. 2006.

[14] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Management in distributed systems: a semi-formal approach," in *Proc. of 13th Intl. Euro-Par 2007 Parallel Processing*, ser. LNCS, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641. Rennes, France: Springer, Aug. 2007, pp. 651–661.

[15] T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, and V. Getov, "Advanced grid programming with components: A biometric identification case study," in *Proc. of the 32nd Intl. Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE, Jul. 2008, pp. 401–408.