Towards hierarchical management of autonomic components: a case study

Marco Danelutto, Marco Aldinucci

`{marcod,aldinuc}@di.unipi.it`

UNIPI

Dept. of Computer Science – University of Pisa

Largo B. Pontecorvo 3, Pisa, Italy

Peter Kilpatrick

`p.kilpatrick@qub.ac.uk`

QUB

Dept. of Computer Science – Queen's University Belfast

University Road, Belfast BT7 1NN, UK.

Towards hierarchical management of autonomic components: a case study

Marco Danelutto, Marco Aldinucci
{marcod,aldinuc}@di.unipi.it
UNIPI
Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

Peter Kilpatrick
p.kilpatrick@qub.ac.uk
QUB
Dept. of Computer Science – Queen's University Belfast
University Road, Belfast BT7 1NN, UK.

Abstract

We address the problem of orchestrating hierarchical autonomic management. We first define the concepts of autonomic manager and QoS contract. Then we consider how general autonomic management strategies can be suitably defined for simple structured parallel programs. We outline how hierarchical manager orchestration can be modelled using Orc and finally we discuss some simple case studies.

1 Introduction

Autonomic management of parallel and distributed computations is a challenging task. It is concerned with those non-functional details (i.e. not directly affecting *what* has to be computed, but rather *how* parallel computation has to be performed) that usually present significant hurdles for application programmers. Recent works tackle the problem of structuring autonomic management of complex, applications [5]. In particular, they relate the Autonomic Computing Reference Architecture (ACRA) with Information Technology Service Management (ITSM) and in so doing they carefully analyse the implications of hierarchical autonomic management as well as the patterns needed to structure such hierarchical management and the interfaces needed to keep management modular and interoperable.

In the framework of parallel/distributed applications, we investigate how hierarchical autonomic management can be organized. We previously introduced the behavioural skeleton concept, a composite component that exposes a description of its functional behaviour and establishes a parametric orchestration schema of its inner components as well [1]. 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. Thus, our work is aimed at contributing to the implementation of fully automatic

systems as proposed in [9]. We first present a general overview of hierarchical autonomic management framework (Sec. 2); then we provide a model of the autonomic managers involved, which can form the basis for reasoning about the overall system behavior (Sec. 4); 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).

2 Autonomic management and contracts

The term *Autonomic Computing* is emblematic of a *hierarchy* of self-governing systems, which may consist of many interacting, self-governing components that in turn comprise a number of interacting, self-governing components at the next level down. 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. 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. 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 [9]. Each manager pursues a goal specified in a *QoS contract* [2]. In the context of autonomic component models, such as GCM [6], 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 categorise 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 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. 1, the management is strictly hierarchic and contracts can be unfolded along a tree, whereas in the 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 geographical locality. Also, the strictly hierarchic structure often reflects the way that IT professionals are organised thus enabling the specialisation of managers for area of concerns [5].

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 components 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 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

---

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

[2]where $\perp$ represents undefined variables (e.g. due to a timeout in their gathering).
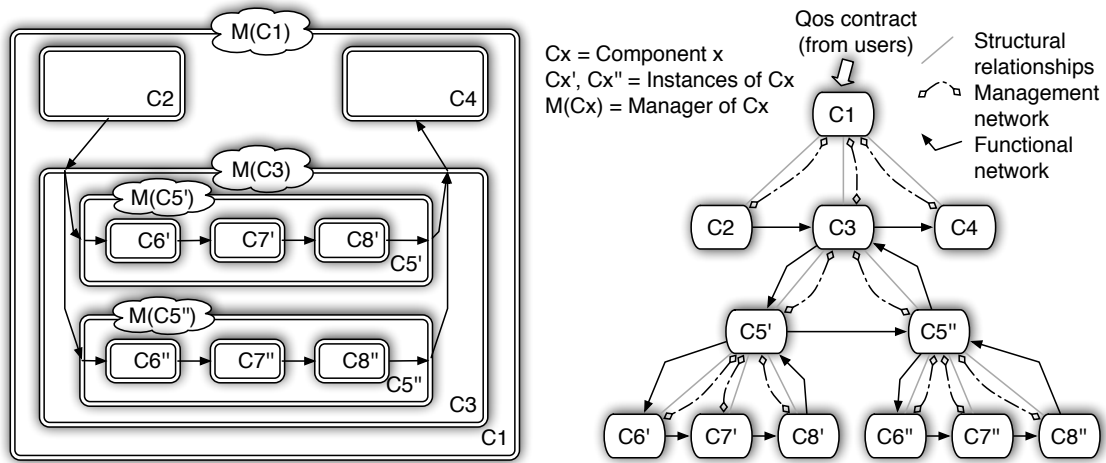
Figure 1: An autonomic component application and its structure. Three relationships between components are highlighted.

$\mathcal{CP}$ 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;

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 deployng a new component instance, sending a message in the proper format (examples can be found in [1]).

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 (e.g. $m_0(\circlearrowleft_{t+3}) = g(m_1(\circlearrowleft_t), m_2(\circlearrowleft_t)), m_1(\circlearrowleft_{t+3}) = f(m_2(\circlearrowleft_t)))$. 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.
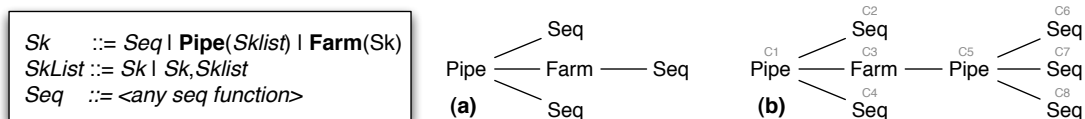
Figure 2: Skeleton grammar (left) and sample skeleton programs (right, shaded identifiers are only here to be able to refer program parts in the text)

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. 2 left (we assume here that "Seq" will be a generic *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 [4]).

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 capitalize 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 [1]. However, the results discussed in the 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 been more difficult.

Typical programs of the skeleton framework considered here are those represented in Fig. 2. 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 parallelization of program (a) if the programmer recognizes 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 [8].

4 Autonomic cycle

We present an Orc [11] 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 [1] 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 [3]: 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: $\gg$ is sequential composition; $> v >$ denotes sequential composition with passing of a value; | 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.

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

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 [3]. 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.

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

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.

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

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

$$adapt(farm(N), plan) \triangleq (\texttt{if}(plan = addworker) \gg let(y) \gg farm(N + 1, contract)$$

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

Table 1: Managers and contracts for program (b) of Fig. 2 (*self* denotes the component itself, *super* denotes the father component).

Component	Manager Contract	m_i
pipe	$\mathcal{CP} = K_{\text{low}} \leq T_{\text{self}} \leq K_{\text{high}}$	$K_{\text{low}}, K_{\text{high}}$ constants; $\forall C_f \in \text{children(self)}$: T_{C_f} monitored (service time); $T_{\text{self}} = \max_{\forall C_f}\{T_{C_f}\}$; $\forall C_f : \mathcal{CP}_{C_f} = \mathcal{CP}$
farm	$\mathcal{CP} = (\mathcal{CP}_{\text{super}}) \wedge (IT_{\text{self}} \leq T_{\text{self}})$	IT_{self} monitored (request inter-arrival time); $\forall C_f \in \text{children(self)}$: T_{C_f} monitored (service time); n_{self} = #workers; $T_{\text{self}} = \sum_{j=1..n} T_{C_j}/n$; $\mathcal{CP}_{C_f} = \text{optimise}(T_{C_f})$; $\mathcal{G} = 1/n_{\text{self}}$

Table 2: Description of the management contracts for farm and pipe behavioural skeletons (*self* denotes the component itself, *super* denotes the father component).

$$\texttt{where} \quad (\forall i :: y_i :\in Interrupt_i.set)$$

The synchronization following the publication of all $Interrupt_i.set$ signals is achieved using the barrier synchronization idea of [11].

5 Use cases

We consider a typical use case and discuss how hierarchical autonomic management proceeds for some typical situations. The results discussed here are not achieved with actual code, but rather by simulating the manager hierarchy as described in Sec. 2 and modelled through the Orc code of Sec. 4. However, the presented managers *taken in insulation* have been developed and experimented in a previous work [1].

We consider program (b) of Fig. 2 and assume the contracts of interest deal with *service time*[3]. The service time for skeleton Sk is denoted by $T(Sk)$.

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 $[K_{\text{low}}, K_{\text{high}}]$, and the general contracts for farm and pipe decribed in Table 2. 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 3.

---
[3]the average time spent between the delivery of the results of two consecutive processing requests.

	Plan	Expected Cost	Expected Benefit
PL_{F_1}	move the slower worker C_w on a faster platform, if any	$h = \text{cost}(\text{stop}(C_w);\text{deploy}(C_w);\text{start}(C_w))$	decrease service time. $T_F(\circlearrowright_{t+h}) = T_{C_x}(\circlearrowright_t)\Delta$, $0 \leq \Delta \leq 1$ speed difference between the platforms
PL_{F_2}	increase parallelism degree (allocate k new workers)	$h = \text{cost}(\text{deploy}(C_x);\text{start}(C_w))$ for k instances	decrease service time. $T(\circlearrowright_{t+h}) = T(\circlearrowright_t)n/(n+k)$
PL_{F_3}	decrease parallelism degree (deallocate k workers)	$h = \text{cost}(\text{stop}(C_w))$ for k instances	increase service time. $T(\circlearrowright_{t+h}) = T(\circlearrowright_t)(n+k)/n$
PL_{F_4}	raise violation		
PL_{P_1}	move stage (C_s) with maximum T to a faster resource, if any	$h = \text{cost}(\text{stop}(C_s); \text{deploy}(C_s); \text{start}(C_s))$	decrease service time. $T_P(\circlearrowright_{t+h}) = T_{C_P}(\circlearrowright_t)\Delta$, $0 \leq \Delta \leq 1$ speed difference between the platforms
PL_{P_2}	collapse fastest adjacent stages (C_s, C_{s+1})	$h = \text{cost}(\text{stop}(C_s); \text{deploy}(C_s); \text{start}(C_s))$ for C_s, C_{s+1}	decrease resource usage $n = n - 1$, increase service time
PL_{P_3}	raise violation		

Table 3: Initial set of plans considered for pipeline and farm managers.

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, PL_{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 new faster resources PL_{P_1} cannot be applied. PL_{P_2} cannot be applied as well because 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_1}, which is eventually implemented ($> (b,p) > adapt(sk1,p) > sk2 >$) and a new autonomic cycle is started.

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 Related Works and Conclusions

The idea of autonomic management of distributed applications is present in several programming frameworks, although in different flavours. ASSIST [2], AutoMate [12], SAFRAN [7], and GCM [6] all include autonomic management features. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids. Though

such programming frameworks considerably ease the development of an autonomic application, 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 run-time uncertainty aspects but, at the same time, decreases the component reuse potential since it further specialises components with application specific management code.

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 specialised 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. Simulation results, derived with the support of the models, show that effective management can be implemented when (as an example) service time is to be autonomically optimized. Overall, this is a further step in the definition/refinement of the behavioural skeleton concept introduced in the context of GCM, the Grid Component Model developed within the CoreGRID NoE and the spinoff GridCOMP STREP projects.

References

[1] 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 D. E. Baz, J. Bourgeois, and F. Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, Feb. 2008. IEEE.

[2] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006.

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

[4] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of 11th Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 761–770, Lisboa, Portugal, Aug. 2005. Springer.

[5] 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*, 46(3):565–681, 2007.

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

[7] 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.

[8] GridCOMP. GridCOMP web page, 2007. http://gridcomp.ercim.org.

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

[10] H. Kuchen. A skeleton library. In B. Monien and R. Feldman, editors, *Proc. of 8th Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 620–629, Paderborn, Germany, Aug. 2002. Springer.

[11] J. Misra and W. R. Cook. Computation orchestration: A basis for a wide-area computing. *Software and Systems Modeling*, 6(1):82–110, Mar. 2006.

[12] 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.