Autonomic management of non-functional concerns in distributed & parallel application programming

Marco Aldinucci
Dept. Computer Science
University of Torino
Torino – Italy
aldinuc@di.unito.it

Marco Danelutto
Dept. Computer Science
University of Pisa
Pisa – Italy
marcod@di.unipi.it

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

Abstract

An approach to the management of non-functional concerns in massively parallel and/or distributed architectures that marries parallel programming patterns with autonomic computing is presented. The necessity and suitability of the adoption of autonomic techniques are evidenced. Issues arising in the implementation of autonomic managers taking care of multiple concerns and of coordination among hierarchies of such autonomic managers are discussed. Experimental results are presented that demonstrate the feasibility of the approach.
Keywords: autonomic management, algorithmic skeleton, design pattern, behavioural skeleton, parallel computing, distributed computing, grid, clouds.

1. Introduction

Massively parallel and/or distributed systems are now available based on three different architectural models: high performance clusters, grids [1] and clouds [2]. To facilitate efficient implementation of parallel and/or distributed applications on these architectures, a range of structured parallel programming methodologies is available, such as those based on parallel design patterns [3] or algorithmic skeletons [4], [5]. It is not realistic to design programs with hundreds to thousands of parallel/distributed activities where each activity represents an independent flow of control with its own code, different from the others. Even when applications targeting this kind of architecture are not explicitly programmed according to a structured parallel programming model, their internal structure is usually an implicit hand-coding of some well-know pattern such as the embarrassingly parallel or the map/reduce one.

The management of non-functional concerns in this kind of application is typically a complex task. Non-functional concerns are those concerns not directly related to the result computed by an application, but rather to the way this result is computed. Examples of non-functional concerns include performance, security, fault tolerance. Management of such concerns usually requires extensive knowledge of the target execution environment and appropriate interaction with the functional code of the application. Non-functional concern management becomes increasingly complex as the target architecture becomes more and more dynamic and heterogeneous, and the features of the target execution environment are progressively hidden from the application programmer (moving from clusters to "invisible grids" [6] and from grids to clouds). This suggests or, in some cases, imposes the adoption of some kind of autonomic management [7] of non-functional features. In all cases where the target architectures features are explicitly and intentionally hidden from the user/programmer, e.g. in clouds, the adoption of autonomic features managing non-functional concerns at run time, i.e. when the maximum amount of information relative to the target execution environment is available, becomes a must.

In addressing the need to accommodate autonomic capability we adopt the IBM blueprint ideas: autonomic management of a component is provided by a dedicated autonomic manager which encompasses all autonomic activities while interacting with the functional core of the component [8]. This provides an attractive separation of concerns and, combined with the well understood scalability properties of algorithmic skeletons, presents a potentially effective means of structuring applications for use in grids and clouds.

The results discussed represent the latest episode in a longer-term programme of work that aims to allow an application programmer using, say, a cloud-based setting, to focus on providing the required functional code while simply providing a contract specifying the limits on the relevant non-functional parameters. It will then be the job of the "system" to manage the application so as to maintain the contract. To date

0. This work has been partially supported by EU FP6 NoE CoreGRID, EU FP6 STREP GridCOMP and Italian FIRB Insyeme projects.

we have introduced the idea of behavioural skeletons bringing together the idea of skeleton programming and autonomic management [9]; presented experimental results demonstrating the feasibility of the idea for a single manager taking care of a single non-functional concern [10]; and described a detailed strategy for the interaction of a hierarchy of managers dealing with a single concern [11]. The contribution of this work is twofold: in section 3.2 we highlight the issues that may arise among autonomic managers responsible for differing (but perhaps, in some way, competing) concerns; and in section 4 we discuss an implementation of the strategies underpinning hierarchical management of a single concern and show experimental results demonstrating the efficacy of the approach.

The results discussed stem from various different experiences in several projects in which we have been involved: the Italian FIRB GRID.it [12] project, in which the programming environment ASSIST was designed, which was the first example of the use of autonomic managers taking care of performance tuning in massively parallel applications [13]; the CoreGRID European Union NoE [14] where a Grid Component Model (GCM) was designed and the concept of behavioural skeleton [10] first introduced; and the GridCOMP EU STREP [15] project, where the GCM reference implementation, including the behavioural skeleton idea, was developed.

2. Functional & non-functional application concerns

Functional concerns in the development and implementation of a parallel/distributed application relate to the results computed by the application, i.e. to *what* is computed by an application, while non-functional concerns relate to *how* these results are computed. Non-functional concerns give rise to many of the hard, error prone problems in parallel/distributed computing. For the purposes of this work, we consider the *kind* of parallel patterns exploited to implement the application to be a functional concern, while all the management issues relating to the exploitation of the parallel patterns (for example, parallelism degree set-up and tuning, dynamic load balancing, adaptation of parallelism exploitation pattern to varying features of the target architecture and/or application) represent non-functional concerns.

Non-functional issues are usually tackled directly at the user/application programmer level using various techniques. The parallelism degree can be fixed, either at compile time or at run time, or can be handled in such a way that it can vary during application execution. Fault tolerance can be supported by introducing appropriate checkpointing code and/or using redundant control in such a way that a limited number of faults can be tolerated. Security usually requires transport level data and code encoding together with some reliable authentication system. Adaptivity requires monitoring facilities to be implemented that allow observation of the actual behaviour of a parallel application as well as mechanisms to support the adaptive strategies used when the monitored behaviour is discovered to be non-compliant with the expected one and appropriate adaptation policies are used.

In an ideal programming scenario, functional concerns should be under user/application programmer control and responsibility. Non-functional concerns should instead be completely handled by the "system"[1] in accordance with the general "directions" provided by the user/programmer via, for example, some kind of Service Level Agreement (SLA). Indeed, this is what already happens in the sequential programming scenario: programmers write code using a high-level programming language and then the HLL compiler, its run time support and the underlying operating system take care of those aspects related to memory management that make efficient the computation expressed by the user through the HLL program. The only possibility the user has to influence memory management is through SLAs directed to either the compiler (e.g. a `register` directive modifying the default allocation of a variable in a C program) or the run time system (e.g. a directive modifying the stack size, such as the `-Xsm nnn` passed to a JVM).

When explicitly handling non-functional concerns, the programmer confronts several distinct problems, in particular:

- **code tangling** The code taking care of the non-functional concerns is often intermingled with code implementing the functional part of the application. This proves a hindrance when debugging and tuning either the functional or the non-functional code. Also, failure to separate functional and non-functional code usually prevents simple reuse of non-functional code in applications whose application domain is different but which adopt the same parallelism exploitation pattern(s).
- **wide knowledge requirements** In order to manage efficiently non-functional concerns, extensive knowledge of the target architecture is needed, which is often only available at run time, as well as knowledge of the techniques used to handle non-functional concerns. Both of these kinds of knowledge differ substantially from the domain-specific knowledge required to develop the functional part

1. by programming tools, compilers, run time supports, operating systems or by some kind of cumulative and coordinated effort by these layers altogether

of the application. Consequently, application programmers must have, and maintain, a broad range of skills to develop effective implementations.

These problems may significantly impair the efficiency achieved by application programmers when dealing with non-functional concerns. A possible solution consists in moving all of the non-functional concerns to the compiler/run time support of some high-level programming environment, while leaving the user/application programmer the duty of formulating some general SLA with which the system may be provided and which describes the non-functional concern goals the user requires to be achieved. This is the approach we propose in adopting the concept of behavioural skeleton with associated autonomic manager. The general idea is to provide application programmers with pre-defined parametric parallel programming patterns with built-in handling of non-functional concerns. These patterns may be implemented using standard static (compile time) and dynamic (run time) techniques. The expected result is twofold: on the one hand, application programmers may concentrate on the domain specific application aspects, i.e. those that lie within their area of expertise. As they are no longer distracted by implementation detail of non-functional aspects, their work may be much more effective. On the other hand, non-functional feature handling is encapsulated and becomes the responsibility of system programmers who can bring to bear their specialized knowledge of target architecture and non-functional techniques.

In the context of cloud computing, application programmers no longer have the possibility to program effective non-functional concern management due to the high "virtualization" of this kind of architecture. Therefore non-functional concerns *must* be managed within the cloud compiler/middleware/RTS. In turn, this means non-functional concerns will be managed by programmers with a better knowledge of the target (virtualized) architecture and thus potentially more effective non-functional concern handling can be achieved.

3. Management of non-functional concerns

To support the management of non-functional concerns we employ the concept of *autonomic manager* (AM). In the context of this work an autonomic manager is an independent activity completely and autonomically managing some specific non-functional concern within an application. The AM is actually a concurrent activity with respect to the main flow of control of the application. AMs are characterized in three distinct dimensions: i) the *concern* they manage ii) the *autonomic policies* they implement and iii) the *degree of cooperation* with other managers (if any) in the same parallel application. In particular, the concern and the degree of cooperation represent orthogonal dimensions in the design space considered here. Therefore, we will consider AMs taking care of a single or of multiple goals, AMs taking care of a goal(s) alone or in a hierarchy of coordinated AMs, and any intermediate combination (see Fig. 1 left).

In this work, we employ classical autonomic managers [16], [17]. These managers execute a control loop. A *monitor* phase triggers an *analysis* phase. If behaviour is not as theoretically expected, corrective actions are *planned* and finally *executed*. Immediately after action execution the control cycle restarts with monitoring.

As an example of a single concern autonomous manager, consider an AM whose goal is performance optimization in a parallel application. Several autonomic policies can be implemented within this AM: initial parallelism degree setup, mapping of parallel activities to processing resources, adaptation of parallelism degree in the event of non-temporary target architecture feature variations (e.g. load increase or decrease), load balancing among the available resources, migration of poorly performing activities to faster execution resources, etc. Some of these activities represent very difficult problems, at least in the general case, even where the AM is the only one in the computation or does not cooperate with other AMs in the same computation. For example, the mapping of parallel activities onto available processing resources is a NP-hard problem in the general case. The complexity of management may be exacerbated when inter-AM cooperation has to be taken into account. Suppose another AM, whose main goal is security, is present in the same application. Policies implemented within this manager may concern the ability to secure data and encrypt communications to/from remote nodes that involve non-private network segments. In this case the mapping of parallel activities to processing resources should not only take into account the network dependent communication costs, but also the fact these costs increase when the related network links are non-private.

Apart from mapping, several other examples of hard problems related to AM policies can be cited, including load balancing, optimal allocation of shared data, etc. One means of reducing AM complexity is to restrict in some way the kinds of parallel computations in such a way that the AM problems become more manageable. Here, we leverage on the fact that efficient massively parallel/distributed computations very often implement well-known parallel/distributed exploitation patterns. Following this approach, we introduced in [9] the concept of *behavioural skeleton* (BS). A behavioural skeleton is a pair $\langle \mathcal{P}, \mathcal{M}_{\mathcal{C}} \rangle$, where $\mathcal{P}$ is a well known parallelism exploitation pattern and $\mathcal{M}_{\mathcal{C}}$ is an AM taking care of a concern $\mathcal{C}$ in the computation of $\mathcal{P}$.

To date we have focused on the functional replication pattern. In a functional replication pattern a number of functionally equivalent computations are performed in parallel to process a stream of input data to produce a stream of output results. By varying the way input tasks are distributed to the available concurrent computations, the way the results are gathered into the output stream and the amount of data shared among the concurrent computations, several distinct parallel patterns can be modeled, including embarrassingly parallel computations on streams (task farm) and data parallel computation (embarrassingly parallel or with state or stencil).

Using BSs we succeeded in reducing the complexity of performance optimization and tuning to a tractable size. In particular, we demonstrated that the parallelism degree of computations implemented using a functional replication BS can be initially set to some "optimal"[2] value and then adapted to take into account different availability (load) of the processing resources used as well as different needs (computational power) of the application [10]. Here we extend the work already performed with behavioural skeletons. In particular, we discuss both multi-concern management and cooperation among managers within a hierarchy and present experimental results in relation to the latter.

3.1. Hierarchical management of a single non-functional concern with BS

In those cases where a hierarchy of relatively independent software modules is used to compose an application, hierarchical management of non-functional concerns can be exploited to achieve better results. In hierarchical management of non-functional concerns, managers are attached to the individual software modules within the hierarchy and therefore themselves constitute a hierarchy. Managers in a higher position in the hierarchy will take more autonomous decisions, while managers in a lower position in the hierarchy will behave in consequence of decisions taken at the higher levels. In particular, the top level manager will receive from the user a *contract* (SLA) specifying the constraints on the parameters within which the application must operate in the context of the given non-functional concern. In turn, each lower level manager will be given a (sub-)contract by its parent. The contract is described in a formalism appropriate to the non-functional concern and represents the target for the autonomic activity.

Two main issues arise when considering hierarchical management of non-functional features:

2. with respect to the efficiency achieved with the processing resources used

$\mathbf{P}_{spl}$ A strategy must be devised that allows splitting of a contract c of a top level manager into a set of sub-contracts $c_1, \ldots, c_m$ to be propagated to the nested managers.

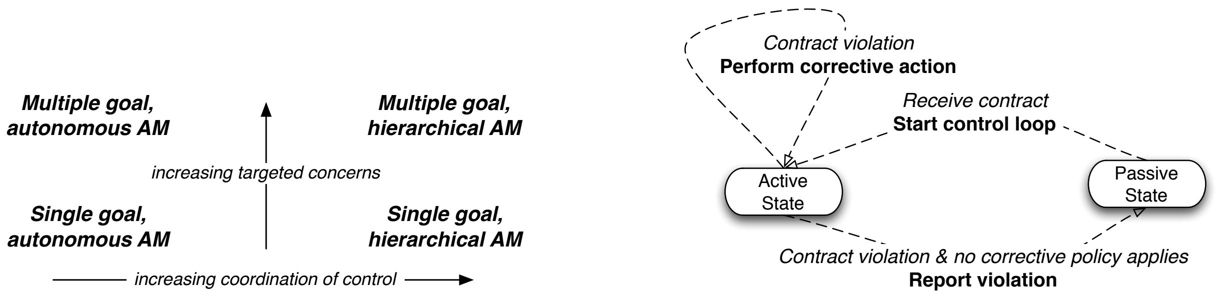
$\mathbf{P}_{rol}$ Each AM should be able to play two independent roles: *active* and *passive*.
In active mode, a manager actively and autonomically tries to ensure the contract received either from the user or from its parent manager in the hierarchy, by executing a classical autonomic control loop and, if required, planning and executing appropriate autonomic actions aimed at ensuring contract satisfaction. In passive mode the manager only monitors the current computation status and waits for a new contract from its parent. Passive mode is entered when an active manager identifies that it no longer satisfies the current contract, for some reason, and there is no (locally available) plan(s) that can be used to recover the situation.

Provided these two issues can be effectively addressed, hierarchical single goal autonomic management can be implemented as follows:

- The user provides a top level contract (application SLA).
- The contract is split into sub-contracts and the sub-contracts are propagated to the children of the top level manager, and the top level manager then enters active mode.
- Recursively, each manager receiving a contract splits it into sub-contracts, propagates them to its children and enters active mode.
- Each manager in the active mode performs a classical autonomic control loop. Where appropriate and possible, corrective actions are taken. If corrective action is required and not possible, a contract violation is reported to the parent and passive mode is entered. The manager remains in passive mode until it receives a new contract.

A large number of massively parallel/distributed applications exploit parallelism through compositions of two basic stream parallel patterns: pipeline and task farm. The former models computations in stages, where computation of stage i on task k may proceed in parallel with computation of stage $i-1$ on task $k+1$ and of stage $i+1$ on task $k-1$. The latter models embarrassingly parallel computations, where computation of task k may proceed in parallel with computation of task k' ($k' \neq k$).

Let us suppose two behavioural skeletons exist, $BS_p = \langle \mathcal{P}_{pipe}, \mathcal{M}_{\mathcal{C}} \rangle$ and $BS_f = \langle \mathcal{P}_{farm}, \mathcal{M}_{\mathcal{C}} \rangle$ modelling pipeline and farms, respectively, and having as concern $\mathcal{C}$. The computations above can thus be modelled with a tree of behavioural skeletons where nodes are BSs and leaves are sequential portions of code representing the lowest level of pipeline stages or task



Multiple goal, autonomous AM
Multiple goal, hierarchical AM
increasing targeted concerns
Single goal, autonomous AM
Single goal, hierarchical AM
increasing coordination of control
Contract violation
Perform corrective action
Receive contract
Start control loop
Active State
Passive State
Contract violation & no corrective policy applies
Report violation

Figure 1. Orthogonal dimensions in autonomic management of non-functional concerns (left) and states of autonomic managers in behavioural skeletons (right).

farm workers.

In this case, the AM associated with the BS at node n in the tree should manage concern $\mathcal{C}$ by coordinating the activities of the descendant node BSs $BS_{n_1}, \dots, BS_{n_m}$ and reporting the results of such coordination to the parent BS BS_{n_0}. These two activities clearly require the interaction of all the involved managers. For example, consider a tree whose pattern structure is given by the expression *farm(pipeline(sequential, farm(sequential), sequential)))* and suppose the non-functional concern is performance optimization. The AM associated with the inner pipeline should optimize performance by interacting with the AMs of its descendant nodes of type sequential, farm and pipeline. Finally, it should report to the AM of the outer, top level farm.

The $\mathbf{P}_{rol}$ problem above can be solved by organizing autonomic management of non-functional features in two distinct parts, a *passive part* implementing the mechanisms needed to monitor the behaviour of the associated computation with respect to concern $\mathcal{C}$ and the mechanisms needed to adjust the behaviour of that computation; and an *active part* implementing the autonomic policies of the AM in such a way that these policies try to maintain the SLA contract received from the user (top level AM) or from the parent AM (inner AMs). This allows separation of policies from mechanisms and ensures that policy formulation can be carried out without consideration of how the policy will be enacted. In Section 4 we will see how this implementation strategy facilitates active and passive role AMs in behavioural skeletons.

The $\mathbf{P}_{spl}$ problem is much more complex. It is difficult to imagine a general strategy for splitting a contract agreed with a particular BS into sub-contracts to be agreed/ensured by the nested behavioural skeletons, nor are we aware of proposed solutions to the more general problem of splitting general purpose SLAs into sub-SLAs that, once ensured, guarantee satisfaction of the original SLA. However, we can adopt quite effective domain specific heuristics once the general contract c is fixed, using the fact that the AMs are associated with well-know parallel patterns. For example, when performance is considered, splitting the SLA of a pipeline can exploit the well-known performance model of a pipeline, in which the pipeline performance is bounded by the performance of the slowest stage. As a consequence, a throughput SLA for the pipeline may be split into identical SLAs for the pipeline stage AMs, and a parallelism degree SLA can be proportionally[3] split into the parallelism degree SLAs of the pipeline stage managers.

3.2. Multi-concern management with BS

When multiple concerns $\mathcal{C}_1, \dots, \mathcal{C}_h$ are considered in the context of autonomic management of non-functional features, further problems arise from the *structuring* of autonomic management activities. Besides hierarchical management structure deriving from nested behavioural skeletons, we have to decide how the management of the different non-functional features is to be coordinated among different managers. At the two extremes lie the following scenarios:

SM a single (hierarchy of) AM exists taking care of the concerns $\mathcal{C}_1, \dots, \mathcal{C}_h$ altogether.

MM multiple (hierarchies of) AMs, each taking care of a different concern $\mathcal{C}_i$ plus a general super-AM orchestrating the multiple AMs so as to take care of $\mathcal{C}_1, \dots, \mathcal{C}_h$.

In both cases[4], the challenge lies in resolving conflicts arising from decisions taken when considering different concerns; or how to derive some kind of "summary" super-contract $\overline{c}$ from $c_1, \dots, c_h$ with its own policies such that managing that contract leads to fair and efficient management of all the concerns $\mathcal{C}_1, \dots, \mathcal{C}_h$.

To illustrate this issue, suppose that performance optimization and security are the non-functional concerns

3. depending on the relative computational weight of the stages
4. and also, in our opinion, in all intermediate cases

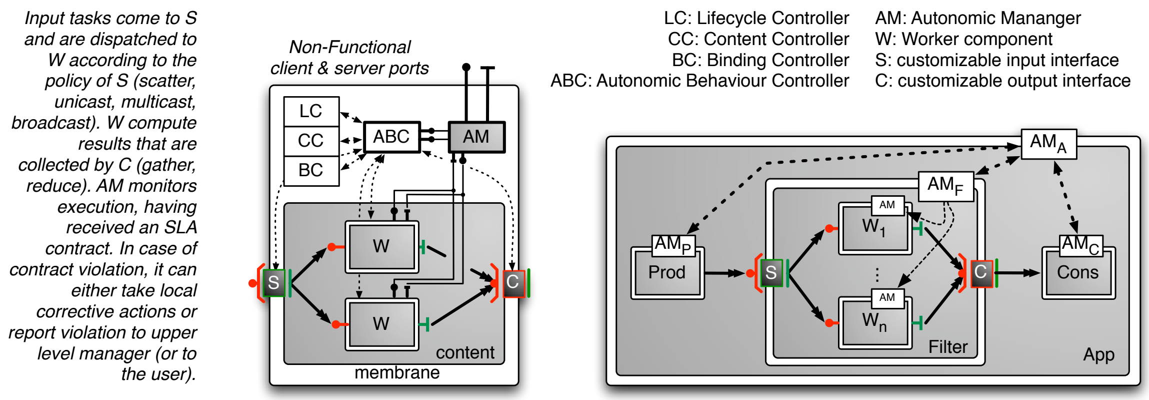

Input tasks come to S and are dispatched to W according to the policy of S (scatter, unicast, multicast, broadcast). W compute results that are collected by C (gather, reduce). AM monitors execution, having received an SLA contract. In case of contract violation, it can either take local corrective actions or report violation to upper level manager (or to the user).

Figure 2. Functional replication BS in GCM (left) and nested usage of BS (right)

$(\mathcal{C}_{perf}, \mathcal{C}_{sec})$ under focus in an application. The application is implemented using a tree of behavioural skeletons whose internal structure is a *pipeline(sequential, farm(sequential), sequential)* (such as the one in Fig. 2 right). Also, assume an SLA comprising two contract goals is agreed with the top level BS: a minimal throughput $\mathcal{T}$ (c_{perf}) and secure communications for all the nodes in domain $untrusted\_ip\_domain\_A$ (c_{sec}). When the contracts are propagated to the pipeline stages it may eventually be discovered that the first sequential stage may satisfy the c_{perf} but, being deployed onto a node in the domain $untrusted\_ip\_domain\_A$, communications must be implemented with SSL instead of plain TCP/IP sockets according to c_{sec} and this, in turn, leads to violation of c_{perf}.

Another typical situation of conflict comes from the pipeline second stage, which is itself a farm. If the farm cannot satisfy c_{perf}, its manager could plan the addition of a new worker[5]. Therefore it recruits a new resource[6] and instantiates a new worker on the resource. As soon as the worker has been started it is included in the farm scheduler and tasks are submitted to the worker for computation. If the recruited resource belongs to domain $untrusted\_ip\_domain\_A$ then a violation of c_{sec} will arise as a result of trying to re-establish c_{perf}.

Now, $\mathcal{C}_{sec}$ is particularly interesting as it represents a *boolean* non-functional concern: data and code communication is either secure or it is not. Therefore, when considering security concerns, they should be given a priority. In both examples above this means that c_{sec} must have priority over c_{perf}. However, this is not enough. Some agreement is needed to perform actions not directly related to security whatever solution we use in the range SM – MM.

5. after verifying the new worker does not require more communication bandwidth than is available

6. possibly interacting with some kind of external resource manager

Consider again the latter example above. Let us assume that a performance manager AM_{perf} and a security manager AM_{sec} are both active and are coordinated through a "root" general manager GM. If AM_{perf} decides to increase the parallelism degree of the farm in the second stage, and the new resource recruited for the purpose belongs to $untrusted\_ip\_domain\_A$, the decision of AM_{perf} cannot be committed without taking into account/informing AM_{sec}. However, informing AM_{sec} is not sufficient in itself. If AM_{perf} actuates the decision by itself, then during the time needed for AM_{sec} to react and direct securing of communications to and from the new (insecure) node, all the communications with the new node will be unsecured. Therefore, some kind of two phase protocol is needed: i) AM_{perf} should express the *intent* to add a new node, ii) AM_{sec} could react by prompting securing of communications and iii) AM_{perf} may then instantiate the new *secure* worker.

In our opinion this problem and the related solutions do not depend on the approach chosen, whether it be SM or MM or one of the intermediate ones. However, adopting a MM approach makes solutions easier to devise due to the complete separation of concerns. AM_{perf} (AM_{sec}) can be designed, implemented and optimized while taking into account $\mathcal{C}_{perf}$ ($\mathcal{C}_{sec}$) alone and there is a reasonable possibility that general purpose policies may be designed for GM independent of the actual $\mathcal{C}_i$ managed by the single AM_j.

This is true also for policies and contracts. For example, in the context of policies, the two phase protocol above is quite general, provided that all managers make available means to ask for contract satisfiability of a given system configuration (e.g. the one with a new worker on the untrusted node) and ways to intervene to finalize the configuration before it is actually used (e.g. imposing secure communication implementation

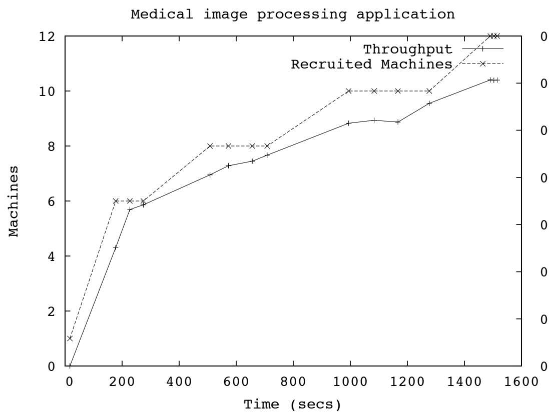


Figure 3. Single AM in action: ensuring a 0.6 task/sec throughput contract/SLA in a task farm BS.

with the new worker). For contracts where non-boolean concerns are considered, it may be possible to devise $\overline{c}$ from $c_1, \ldots, c_h$ using some sort of linear combination. This is an area which requires significant further investigation.

4. Experiments

In this section we discuss experimental results that demonstrate the feasibility and the efficacy of the proposed approach. The results presented here been achieved in the framework of the GridCOMP project [15], where a Grid Component Model (GCM, [18], [19]) reference implementation has been developed. Using this framework we demonstrated previously that autonomic managers can be used to handle non-functional concerns in massively parallel/distributed computations "in isolation", that is, with a single, monolithic manager taking care of a given non-functional feature in the whole application. In particular, we considered separately performance [10] and security issues [20]. Later, we outlined strategies for the realization of hierarchical management of structured parallel computations without presenting an implementation of hierarchical BS or any experimental results [11]. Here we show how hierarchical management of a single concern has been implemented using GCM.

4.1. Behavioural skeleton-based autonomic framework in GCM

A prototype implementation of behavioural skeletons is provided by the GCM reference implementation developed in GridCOMP. Behavioural skeletons are implemented as GCM composite components and provide autonomic management of non-functional concerns in computations whose parallel structure is expressed by an algorithmic skeleton [4]. In particular, autonomic management is provided by the BS autonomic manager (AM) that acts to ensure/restore a user-defined SLA (contract). As outlined in Fig. 2, left, the AM is a membrane[7] component (i.e. a component providing non-functional services). The AM interacts with (uses services provided by) an Autonomic Behaviour Controller (ABC) that provides methods to access the computation status (monitoring) and to implement the actions ordered by the AM (actuators). The ABC, in turn, uses services from the GCM/Fractal standard controllers Lifecycle, Content and Binding Controller to implement both monitoring and actuators.

The autonomic managers implemented in GCM behavioural skeletons use a JBoss rule engine [22] to implement the autonomic control cycle. JBoss rules are *precondition-action* rules. Preconditions are first order formulas over the parameters monitored by the ABC controller. Actions are calls to one or more of the actuator services implemented, again, by the ABC. The control loop itself invokes the JBoss rule engine periodically. At each invocation, "fireable"[8] rules are selected, prioritized and executed. Execution of a JBoss rule leads to the invocation of the actuator mechanisms in the *action* part of the rule, thus affecting the algorithmic skeleton computation running in the BS [23].

Figure 3 plots typical behaviour observed when using a single BS to implement a medical image processing application. The BS used here implements a task farm. Its autonomic manager takes care of performance optimization/tuning. The (user provided) contract specifies that 0.6 images per second be processed and the figure plots the initial set-up of the task farm with the addition of more and more processing resources up to the point where the contract is eventually satisfied. In [10] we have shown how contract satisfaction is guaranteed in the case of changes in either the processing elements used (overload or underload) or in the case of temporary hot spots in image processing.

4.2. Hierarchical management with GCM behavioural skeletons

Following initial experiments with a single manager taking care of a single non-functional concern, we investigated how hierarchical management of non-functional concerns could be implemented, using the approach described in Section 3.1. The AM component in the behavioural skeleton has been extended in such a

7. the concept of "membrane" is inherited from the Fractal component model [21] which constitutes the basis of the reference implementation of GCM in GridCOMP

8. those whose precondition holds true

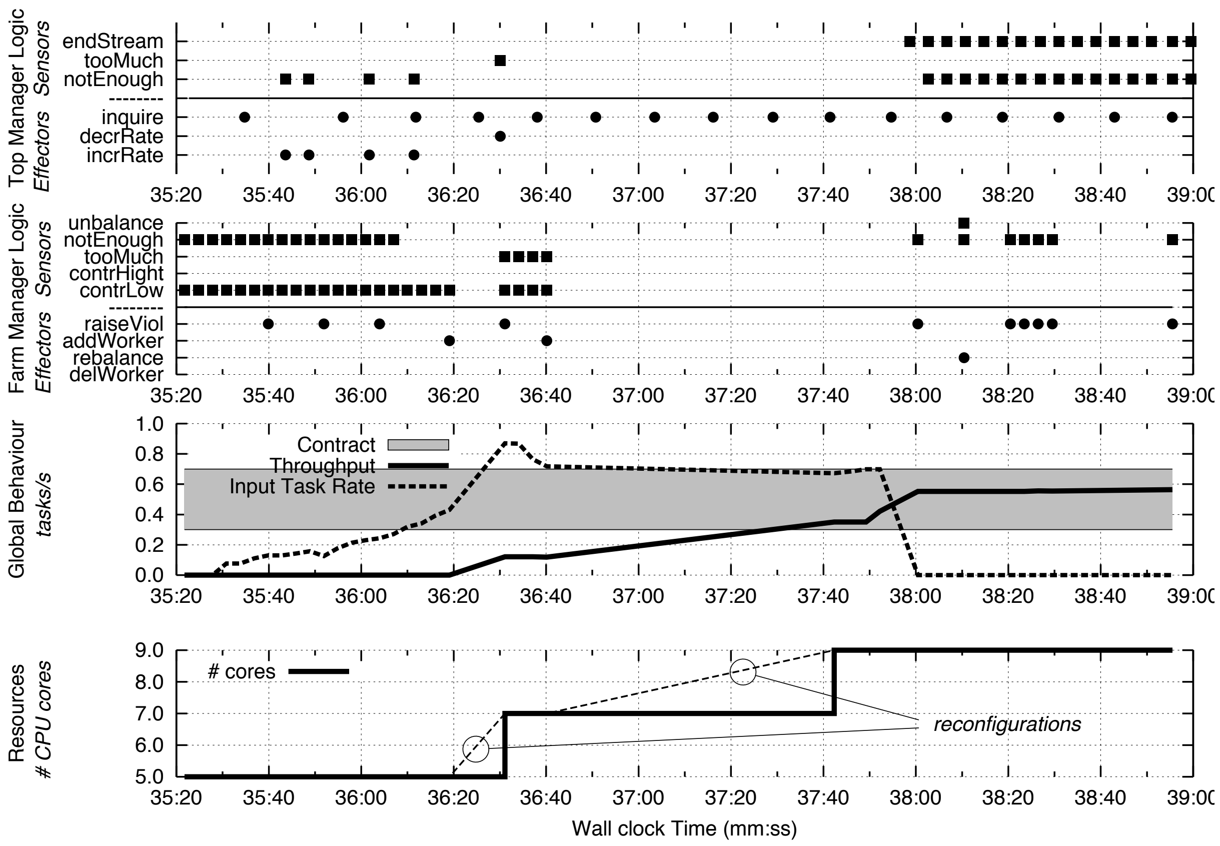


Figure 4. Hierarchical AM in action: actions taken by a task farm BS AM in a three stage pipeline.

way that, besides providing methods to have a contract assigned (active behaviour) and to report monitoring data (passive behaviour), it also provides facilities to report violations to its parent manager. Essentially this involved addition of callback interfaces to signal violations. These new non-functional interfaces are called in the parent manager when the local manager detects a contract violation but cannot apply any rule (among those stored in the JBoss engine database) leading to local actions that could (concur to) re-establish the contract.

The second modification concerned internal AM component policies. The policies are stored as JBoss rules. To model active/passive mode behaviour we simply added rules that, in the event that every other local rule fails, report contract violation to the parent manager. Transition to the passive state is modelled by the absence of fireable "active" rules (rules not raising a violation) in the JBoss database of the AM.

All the modifications have been implemented in the GridCOMP GCM reference implementation. The BS implementation is open source and can be downloaded by following the links at the project web site [15].

Using this extended version of the autonomic managers, and consequently the improved version of behavioural skeletons, we built an application whose structure is depicted in Fig. 2, right. The application is a three stage pipeline. The first stage produces data that are processed in parallel by the second stage and are eventually displayed by the third stage. Four managers are used: AM_A is the manager of the pipeline BS which is the topmost BS in the application (therefore this is the Application AM). AM_P, AM_F and AM_C are the managers of the first (sequential) stage, the Producer; of the second (task farm parallel) stage, the Filter; and of the third (sequential) stage, the Consumer, respectively.

The hierarchy of managers takes care of performance concerns ($\mathcal{C}_{perf}$ in the notation of previous sections).

Initially, a performance (actually, throughput) contract c_{tRange} is passed by the application user to the topmost manager AM_A stating that tasks have to be processed at a rate in the range 0.3 – 0.7 tasks/sec. As the topmost behavioural skeleton is a pipeline, its manager AM_A simply forwards the contract to the stage managers AM_P, AM_F and AM_C. In turn, AM_F, which is a farm manager, does not use the received contract to provide contracts to the worker managers. Rather, it passes the AM_{W_i} a $c_{bestEffort}$ contract in accordance with the definition of task farm BS (see [23]). As a consequence, the AM_{W_i} are effectively in passive mode from the AM_F viewpoint, but in fact they autonomically try to provide the best performance possible locally.

Figure 4 plots what happens during an execution of the program[9] due to the activity of the autonomic manager hierarchy.

The first graph plots events and actions happening in the top level manager AM_A (the pipeline skeleton manager). The second graph plots events and actions happening in the farm manager AM_F. The third graph plots input task rate and throughput of the second pipeline stage, together with the stage's contract. Finally, the last graph plots the resources used to compute the application. A default parallelism degree is set up for the farm stage. The farm resources plus the resources needed to run the producer and consumer pipeline stages amount to a total of 5 cores, initially. The times on the x-axis are wall clock times, in minutes:seconds format. The y-axes represent events in the first two graphs, throughput (tasks per second) in the third one and number of cores used in the fourth one.

During a first phase, AM_F detects that it is not delivering the contract throughput (contrLow event, performance lower than the required contract), but it identifies that this is due to the fact that input pressure (i.e. the rate of arrival of input tasks) is not sufficient (notEnough events, insufficient tasks to keep employed the resources allocated). Therefore, rather than taking any kind of corrective action, it reports a contract violation (raiseViol event) to the upper manager (AM_A) and enters passive mode.

AM_A, upon receiving the first violation event from AM_F (a little bit after time 35:40 because of the network and run time support overheads), sends a new contract to AM_P (incRate event). The new contract demands an increase in its output rate. At time 36:10 the first stage starts to deliver as many tasks as are needed to ensure the contract in AM_F (input task rate within the contract stripe in the third graph). Actually, because of the multiple incRate actions in AM_A, the first stage produces tasks more and more frequently.

In a second phase, AM_F determines that there is still a violation of the contract (contrLow after 36:10), but this time one which can be locally managed: AM_F can increase the farm parallelism degree as now there are sufficient tasks on the input to feed more workers. Therefore it starts adding two new workers (addWorker event around time 36:20) and immediately after (at 36:30) the new workers start processing incoming tasks and the throughput increases. No sensor data is available for AM_F during the reconfiguration, i.e. the second graph reports no data from 36:20 to 36:30. Immediately after that, the AM_F detects that now too many input tasks per second are arriving with respect to the number needed to fulfil the contract c_{tRange} and it, in turn, reacts by asking the first stage of the pipeline (the task producer) to slightly decrease the output rate; this means sending a new contract to AM_P (decRate event). This type of violation is a warning and is conceptually different from the previous one because (strictly speaking) it is useless to enforce the contract. In fact, when too many tasks are arriving the farm can buffer them or transiently delay the reception of messages (buffer full). However, if the farm is configured for "unlimited buffering", this kind of action helps in fine tuning the farm memory usage and is therefore useful in the realization of dynamic self-management strategies.

At time 36:30 AM_F again identifies that the contract is still not satisfied and since the inter-arrival rate of tasks is sufficient (contrLow and !notEnough farm events) it therefore starts to add (two) new workers (time 36:40, new addWorker event). This time reconfiguration takes a little bit longer due to the higher number of components (workers) involved, but eventually the throughput rate required by the contract c_{tRange} is achieved.

In the last phase (rightmost part of the graphs) all the tasks are in the input queues of the workers[10] as denoted by the endStream events in the AM_A (first graph). In this state, the AM_A stops reacting to notEnough events received from AM_F since it cannot take any significant action; since no compensating actions are taken by the AM_F, the event notEnough will persist in time in the event line. At the same time (e.g. at 38:10), the AM_F (second graph) locally reacts to an unbalancing event (rebalance) to redistribute queued input tasks in a fairer way among the workers. Figure 5 lists the rules (in the JBoss source syntax) programmed in the AM_F to implement this case study.

This experiment demonstrates that the approach discussed in section 3.1 is feasible. Autonomic adaptation has also been achieved in the case of additional (external) load upon the cores used for the computation of the BS application. In this case, overloaded workers (pipeline stages) began to deliver fewer results than expected and the manager reacted by adding workers to the farm (in the pipeline stage case we are investigating ways to transform the pipeline stage into a farm with the workers behaving as instances of the original stage).

Overall, this demonstrates that autonomic actions can deliver dynamic adaptation of the computation in

9. the program was run on an 8-core (dual quad-core) SMP machine running CentOS Linux with kernel 2.6.18

10. the ProActive Active Objects used to implement managers and workers use asynchronous communication primitives

```
rule "CheckInterArrivalRateLow"
  when
    $arrivalBean : ArrivalRateBean( value < ManagersConstants.FARM_LOW_PERF_LEVEL)
  then
    $arrivalBean.setData(ManagersConstants.notEnoughTasks_VIOL);
    $arrivalBean.fireOperation(ManagerOperation.RAISE_VIOLATION);
end

rule "CheckInterArrivalRateHigh"
  when
    $arrivalBean : ArrivalRateBean( value > ManagersConstants.FARM_HIGH_PERF_LEVEL)
then
    $arrivalBean.setData(ManagersConstants.tooMuchTasks_VIOL);
    $arrivalBean.fireOperation(ManagerOperation.RAISE_VIOLATION);
end

rule "CheckRateLow"
  when
    $departureBean : DepartureRateBean( value < ManagersConstants.FARM_LOW_PERF_LEVEL )
    $arrivalBean : ArrivalRateBean( value >= ManagersConstants.FARM_LOW_PERF_LEVEL )
    $parDegree: NumWorkerBean(value <= ManagersConstants.FARM_MAX_NUM_WORKERS)
  then
    $departureBean.setData(ManagersConstants.FARM_ADD_WORKERS);
    $departureBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
    $departureBean.fireOperation(ManagerOperation.BALANCE_LOAD);
end

rule "CheckRateHigh"
  when
    $departureBean : DepartureRateBean( value > ManagersConstants.FARM_HIGH_PERF_LEVEL )
    $parDegree: NumWorkerBean(value > ManagersConstants.FARM_MIN_NUM_WORKERS)
  then
    $departureBean.fireOperation(ManagerOperation.REMOVE_EXECUTOR);
    $departureBean.fireOperation(ManagerOperation.BALANCE_LOAD);
end

rule "CheckLoadBalance"
  when
    $VarianceBean : QuequeVarianceBean( value > ManagersConstants.FARM_MAX_UNBALANCE)
  then
    $VarianceBean.fireOperation(ManagerOperation.BALANCE_LOAD);
end
```

Figure 5. Sample JBoss rule file: rules used in the AM_F manager of Figure 2 (right), at work in Figure 4

such a way that the given contract is ensured. It also demonstrates, as a side effect, that BS design is an effective design: the clear separation of autonomic management and parallelism exploitation concerns allows the adoption of very effective solutions. These solutions comprise the rules in the JBoss engines implementing AM control together with the actions executed upon rule firing, i.e. the actuators provided by the ABC controller. The results obtained are far beyond those that can be achieved by a classical algorithmic skeleton run time support.

5. Related work

In [24] the authors discuss how the JBoss rule engine can be used to implement self-healing non-functional concerns in web services orchestrated through BPEL. Two distinct languages are introduced to program the active and passive part of the manager, roughly corresponding to the AC and AM modules in the behavioural skeletons. There are several similarities with our work but their approach appears to be more domain specific. [25] discusses in general the implementation of self-adaptive strategies aimed at ensuring given goals. The work is heavily related to agent technology, as are most of the papers on the subject, and do not address the non-functional concern problem as a whole. [26] and [27] present examples of hierarchical autonomic management where autonomic managers cooperate to achieve a common (non-functional) goal. In the former power consumption is the target concern for autonomic managers. Mathematically well-founded optimizations are implemented in the managers, but the overall goal is quite narrow and it is unlikely the results could be easily generalized to the management of other typical non-functional concerns. In the latter, performance tuning is handled but only by relying on load redistribution among nodes where uneven load balance is observed. [28] discusses performance self-adaptation in grid environments and has many points of contact with our work. However, the authors do not assume the existence of some abstract performance model to be matched, and only rely on observing "low level" features relative to the execution of the application on the target environment to take the corrective actions implementing the self-adaptation process. Jade is a component based autonomic framework that separates the level of managers from the level of managed computations [29] in much the same way as is done in ASSISTANT [30]. Both are examples of how non-functional concern management is perceived as one of the more significant aspects in distributed/parallel computing, as we do in this work.

6. Conclusions

In this paper we outlined problems relating to the handling of non-functional concerns in parallel/distributed computing. In particular, we discussed the main issues arising when multiple concerns come into play simultaneously; and when a hierarchical structure of components, each with its own manager, is present. We presented an approach in which parallel exploitation patterns are combined with autonomic managers to provide effective management of non-functional concerns while retaining scalability. In the experiment section we reported results that demonstrate the efficacy of the proposed methodology in the case of hierarchical management of performance optimization/tuning in structured parallel computations implemented via behavioural skeletons. In previous work we reported preliminary experiments on the cost of handling security concerns in structured parallel computations such as those modelled by BS hierarchies [31]. We have also explored the autonomic management of security in structured parallel computations. In particular, we have investigated the possibility of determining, in an autonomic way, whether code staging and data communications have to be performed using a secure protocol, based on meta-information describing the security of the network interconnections used [20]. The proposed strategy ensures the use of secure protocols only when strictly needed, thus avoiding the introduction of unnecessary overheads involved in code and data encryption and decryption. We are currently building on this experience to implement autonomic management of security concerns in the BS framework. Following this, we will attempt to formulate more precisely strategies for the combined management of security and performance concerns based on the ideas outlined in Sec. 3.2.

Acknowledgements

We wish to thank the anonymous referees for their helpful comments and suggestions. We wish also to thank S. Campa, P. Dazzi, N. Tonellotto, M. Vanneschi and G. Zoppi all of whom contributed in different ways to the work discussed in this paper.

References

[1] I. Foster and C. Kesselmann, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Dec. 2003.

[2] A. Weiss, "Computing in the clouds," *netWorker*, vol. 11, no. 4, pp. 16–25, 2007.

[3] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison-Wesley Professional, 2005.

[4] M. Cole, "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Usenix OSDI '04*, Dec. 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

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

[7] A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal - Autonomic Computing*, vol. 42, no. 1, pp. 5–18, 2003.

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

[9] 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, Jun. 2007.

[10] 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.

[11] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Towards hierarchical management of autonomic components: a case study," in *Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing*. Weimar, Germany: IEEE, Feb. 2009, to appear.

[12] *The GRID.it home page*, 2007. [Online]. Available: http://www.grid.it

[13] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi, and C. Zoccolo, "Self-configuring and self-optimizing grid components in the GCM model and their ASSIST implementation," in *Proc of. HPC-GECO/Compframe (in conjunction with HPDC-15)*. Paris, France: IEEE, Jun. 2006, pp. 45–52.

[14] *The CoreGRID web site*, 2007. [Online]. Available: http://www.coregrid.net

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

[16] *IBM Autonomic Computing home page*, IBM Research, 2005. [Online]. Available: http://www.research.ibm.com/autonomic/

[17] 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.

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

[19] *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

[20] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Adding metadata to Orc to support reasoning about grid programming," in *Towards Next Generation Grids (Proc. of the CoreGRID Symposium 2007)*, ser. CoreGRID, T. Priol and M. Vanneschi, Eds. Rennes, France: Springer, Sep. 2007, pp. 205–214.

[21] E. Bruneton, T. Coupaye, and J.-B. Stefani, *The Fractal Component Model, Technical Specification*, ObjectWeb Consortium, 2003. [Online]. Available: http://fractal.objectweb.org/specification/

[22] *JBoss rules home page*, Red Hat Middleware, 2008. [Online]. Available: http://www.jboss.com/products/rules

[23] M. Aldinucci, M. Danelutto, G. Zoppi, and P. Kilpatrick, "Advances in autonomic components & services," in *From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008)*, ser. CoreGRID, T. Priol and M. Vanneschi, Eds. Las Palmas, Spain: Springer, Aug. 2008, pp. 3–17.

[24] L. Baresi, S. Guinea, and L. Pasquale, "Self-healing bpel processes with dynamo and the jboss rule engine," in *ESSPE '07: International workshop on Engineering of software services for pervasive environments*. ACM, 2007, pp. 11–20.

[25] M. Morandini, L. Penserini, and A. Perini, "Towards goal-oriented development of self-adaptive systems," in *SEAMS '08: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2008, pp. 9–16.

[26] N. Kandasamy, S. Abdelwahed, and M. Khandekar, "A hierarchical optimization framework for autonomic performance management of distributed computing systems," in *ICDCS '06: Proc. of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE, 2006, p. 9.

[27] B. Khargharia, S. Hariri, and M. S. Yousif, "Autonomic power and performance management for computing systems," *Cluster Computing*, vol. 11, no. 2, pp. 167–181, 2008.

[28] G. Wrzesinska, J. Maassen, and H. E. Bal, "Self-adaptive applications on the grid," in *PPoPP '07: Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2007, pp. 121–129.

[29] *Jade home page*, Sardes Project, 2008. [Online]. Available: http://sardes.inrialpes.fr/jade.html

[30] R. Fantacci, D. Tarchi, C. Bertolli, G. Mencagli, and M. Vanneschi, "Next generation grids and wireless communication networks: towards a novel integrated approach," *Wireless Communications and Mobile Computing*, vol. 8, pp. 1–23, 2008.

[31] M. Aldinucci and M. Danelutto, "The cost of security in skeletal systems," in *Proc. of Intl. Euromicro PDP 2007: Parallel Distributed and network-based Processing*, P. D'Ambra and M. R. Guarracino, Eds. Napoli, Italia: IEEE, Feb. 2007, pp. 213–220.