Managing Adaptivity in Parallel Systems

Marco Aldinucci[2,⋆], Marco Danelutto[1], Peter Kilpatrick[3],
Carlo Montangero[1], and Laura Semini[1]

[1] Dept. of Computer Science, Univ. of Pisa
[2] Dept. of Computer Science, Univ. of Torino
[3] Dept. of Computer Science, Queen's Univ. of Belfast
aldinuc@di.unito.it, p.kilpatrick@qub.ac.uk,
{marcod,monta,semini}@di.unipi.it

Abstract. The management of non-functional features (performance, security, power management, etc.) is traditionally a difficult, error prone task for programmers of parallel applications. To take care of these non-functional features, autonomic managers running policies represented as rules using sensors and actuators to monitor and transform a running parallel application may be used. We discuss an approach aimed at providing formal tool support to the integration of independently developed autonomic managers taking care of different non-functional concerns within the same parallel application. Our approach builds on the Behavioural Skeleton experience (autonomic management of non-functional features in structured parallel applications) and on previous results on conflict detection and resolution in rule-based systems.

1 Introduction

When designing, implementing and debugging parallel applications a number of non-functional concerns typically have to be taken into account and properly managed. A *non-functional* concern (sometimes referred to as *extra* functional concern and more recently referred to as *quality attribute*) is a feature not directly affecting *what* the parallel application computes, that is the parallel application result. Rather, it is a feature affecting *how* the parallel application result is computed. Notable examples of non-functional concerns in parallel applications are performance, fault tolerance, security, power management, with performance often being the most important.

Properly managing a non-functional concern usually requires the design, implementation and tuning of code additional to that needed to compute the results of a parallel application (the so-called *business code*). The kind of code needed to manage a non-functional concern poses additional requirements on the application programmer, as correct management of non-functional concerns usually requires a quite deep understanding of the target architecture, which is not usually required when writing business code (only). As an example, performance

⋆ This work has been partially supported by EU FP7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems".

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 199–217, 2013.
© Springer-Verlag Berlin Heidelberg 2013

optimization requires a clear vision of target architecture features in order to be effective. Moreover, the non-functional code is often deeply interwoven with the business logic code, thus resulting in much more difficult debugging and tuning of both business logic and non-functional concern management code.

Radically different approaches may be taken to manage non-functional concerns if we recognize that non-functional concern management is a completely independent activity w.r.t. business logic (functional code) development. In fact, non-functional concern management can be organized as a policy insurance procedure piggy backed onto business logic code. The policies used while managing non-functional concerns are the *non-functional programs* and the mechanisms used to implement these programs–typically those mechanisms used to "sense" the computation status and to "actuate" policy decisions–represent the *assembler instructions* of non-functional management.

If this perspective is taken, then management of non-functional concerns may be implemented as an autonomic engine associated to the business logic code. We can implement MAPE (monitor, analyze, plan, execute) loop based managers where monitoring and execution of actions–those devised by policies in the analyze phase and planned by other policies in the planning phase–happen through the sensor and actuator mechanisms provided by the non-functional concern management assembly instructions. Fig. 1 outlines this general idea.

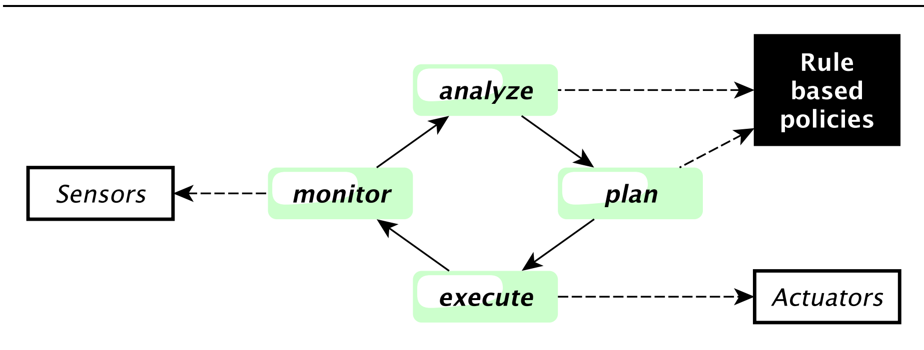

Fig. 1. MAPE loop in autonomic management: the control flow is represented by solid arrow lines and the dependencies are represented by dashed lines

Previous work has demonstrated the feasibility of such an approach to non-functional concern management [2,3,18,1]. These works also pointed out two interesting and somehow conflicting facts:

- Best management policies may be provided by experts in the specific non-functional concern, rather than by "general purpose" non-functional concern experts and/or application programmers.
- Different non-functional concern management policies may lead to conflicts, that is decisions in relation to management of non-functional concern A may impair decisions in relation to management of non-functional concern B.

Therefore autonomic management of different non-functional concerns poses an interesting problem: how can we "merge" policies managing different non-functional concerns without incurring serious penalties due to policy conflicts?

The rest of the paper introduces non-functional concerns and their autonomic management in more detail (Sec. 2 and Sec. 3). Then conflict detection techniques are introduced (Sec. 4). Sec. 5 discusses formal support for policy merging and presents experimental results to assess the complete methodology. Finally, conclusions are drawn in Sec. 6.

2 Non-functional Concern Management in Parallel Computing

As stated in Sec. 1, a non-functional concern is a feature related to *how* the results of an application are computed rather than to *what* these results actually are. Typical non-functional concerns in parallel applications include:

Performance. By far, the most significant non-functional concern in parallel programming. Usually, two distinct kinds of optimization may be required, either latency or service time optimization, with differing implications for the pattern used to exploit parallelism.

Security. Security requirements may be related to data processed and/or to the code used to process input data to get output results. These requirements may vary depending on the kind of resources used to compute the parallel application: shared, private, reserved (i.e. not private, but with exclusive access guaranteed).

Fault Tolerance. Considering the number of resources involved in large-scale parallel applications, it is quite common to experience hardware faults during the execution of an application. Thus fault tolerance is particularly critical to ensure correct completion of applications in the event of failure of (part of) the resources used, especially in the case of long-running applications.

Power Management. If different resources are available (with differences both in terms of power consumption and of performance delivered) power saving becomes a fundamental option in parallel processing, especially at large/extreme scale.

In most cases, the management of these non-functional concerns requires quite complex activities, including:

- Adoption of more complex mechanisms and tools with respect to those needed to support business code only. For example, to ensure security, SSL connections may be required instead of plain TCP/IP connections.
- Parallelization of sequential code or further parallelization of parallel code. For example, in a data parallel computation the input data should be partitioned among a larger number of threads to ensure a shortened completion time of the application. Or the presence of a sequential bottleneck in a parallel computation may require parallelization of the bottleneck code.

– Complete restructuring of the parallel application, i.e. changing the parallel design pattern used to exploit parallelism in the application. For example, having first used a stream parallel pattern, we may realize that the performance of our parallel application is not sufficient and may therefore apply some data parallel pattern also on the different stream parallel pattern components.

In general, various policies may be adopted to deal with non-functional concerns, with different applicability pre-conditions and different results. For example, when dealing with performance, if an application is not performing as expected when running on a heterogeneous architecture, we can either try to move parallel computation components of the application to more powerful architecture nodes (processing elements) or we can try to improve the "structure" of the parallel application (e.g. by changing the parallel pattern used) to give better performance on the existing and available computing resources.

It is worth pointing out that, in general, it is easier to devise suitable management policies when the structure of the parallel application is completely exposed. If the structure is not exposed, it is much more difficult to determine what exactly is going on and thus to plan corrective action in the event of a (non-functional) malfunction of the application. Indeed, without a general view of the application parallel structure, it may even be difficult to realize that there is a non-functional malfunction.

If the parallel pattern of the application at hand is completely exposed we are enabled:

1. to verify whether the application is performing as expected, as the (parallel) design pattern used will come with models that can be verified while the application is running; and
2. to take the decisions suggested by the design pattern used to correct possible problems/malfunctions.

Of course, the parallel pattern–or the pattern composition–used within the application may be identified in two distinct ways: i) by analyzing the HLL (High Level Language) code used to program the application (e.g. where we use a programming framework based on algorithmic skeletons), or ii) by running some kind of (data flow) analysis on the application code to determine whether the underlying parallel activities fit one of the known parallel patterns.

3 Autonomic Management of Non-functional Concerns

Autonomic managers of non-functional concerns may be programmed as outlined in Sec. 1 using MAPE loops. A MAPE loop is a control loop cycling on four different phases (see Fig. 1):

M A *monitoring* phase, where the current status of the running parallel application is observed by collecting data on what happens on the actual target architecture: how many (partial) results have been computed, the time spent

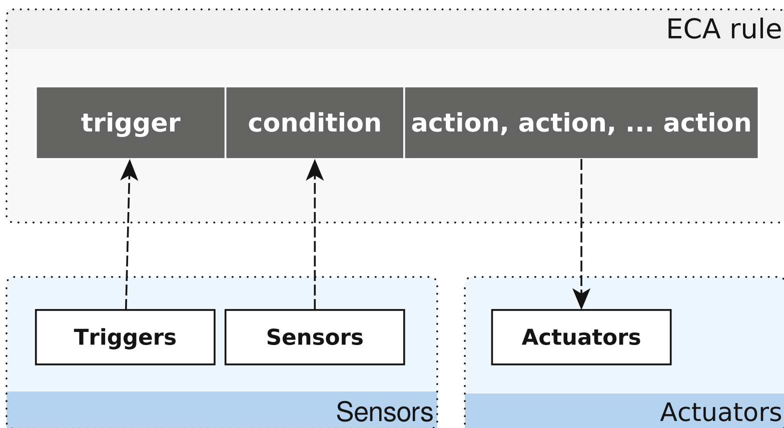

Fig. 2. Implementation of ECA rules

in the different running tasks, the amount of resources used (CPU, Memory, Network), etc.

A An *analyze* phase, where the current situation is analyzed, the behaviour of the parallel application is compared to the expected behaviour and, possibly, a plan to improve application behaviour is selected.

P A *plan* phase, where the decisions taken in the analyze phase are turned into a sequence of actions to be run on the current application.

E An *execute* application, where the plan is actually executed.

The monitoring and plan+execute phases rely on the existence of a set of mechanisms with the ability to "sense" application behaviour and to "act upon" application execution, i.e. to apply the plans devised by the manager policies in the analyze+plan phases. These mechanisms–sensors and actuators–represent "passive" code, as they are just called from within the manager. They also represent *de facto* the interface of the autonomic manager with the (running) business code of the application and determine the kind of policies that can be effectively implemented in the manager.

To clarify the concept, consider an application whose parallel pattern is based on the master/worker paradigm. The availability of sensors reporting to the monitoring phase the number of workers executing and the service time delivered by the master/worker combination determines the capability to react to poorly performing application states. In the same way, the availability of sensors capable of reporting whether a parallel application component is running on a private or on a public resource will enable the manager policies to take correct decisions to ensure application security. On the other hand, the existence of mechanisms

(actuators) capable of stopping and restarting the application, recruiting new resources and deploying and starting active code on the these newly acquired nodes is fundamental to implementation of smart management policies, such as increasing the number of workers in the master/worker pattern or moving an application component from a public node to a private/reserved node.

As far as the "active" part of the MAPE loop is concerned–the analyze and plan phases–different choices can be made. Plain code can be used to hard-wire policies and plans and to call the sensor and actuator mechanisms. However, if we wish to experiment with different policies, or investigate changing policies "on-the-fly" depending on the perceived application status, a more dynamic and declarative solution is necessary. Various systems, including the authors' Behavioural skeletons [2,3], use ECA (Event Condition Action) rule systems to implement manager policies.

An ECA rule is applied in a context that consists of the status of the system at the beginning of the MAPE cycle and the set of events that occurred in the previous cycle. Events may be *external*, that is generated in the system environment, or *internal*, that is generated as part of the effect of an action performed in the previous cycle. The application of a rule results in a new state and possibly in (internal) events, to be considered in the next cycle, when also the external events received in the current cycle will be considered.

More precisely, an ECA rule is a triple

$$\langle trigger, condition, action \rangle$$

Whether a rule is applied in a MAPE cycle depends on its trigger and condition. The trigger is a pattern describing of the events that may cause the application of the rule (a.k.a. *firing* of the rule). At the beginning of the cycle, the trigger is matched with the events in the current context. In case of success, the rule becomes ready to fire: it actually fires, that is, its action is executed, only if its condition holds in the current state. An event that is not matched, or is matched when the condition does not hold, is lost. A single event can enable two or more rules if it matches their triggers. In the case two or more rules are enabled in an evaluation step they are fired concurrently.

The matching process may bind parameters in the trigger to the values carried by the event: the scope of such binding covers the rule condition and action, thus enhancing the capability of the notation to express complex policies. For the same purpose, two triggers may be disjoint, meaning that either is sufficient to apply a rule, and conditions may be combined with the standard logical operators.

Fig. 2 shows how selectors and actuators can enact ECA rules.

The use of ECA rules allows a better implementation of the manager policies. In particular, we use the triggering event to start rule evaluation. In previous work, we used JBoss rule syntax to express management rules. In that case rules were tested cyclically for fireability. The period of the cycle *de facto* determined the MAPE loop efficiency, as "too slow" loops react poorly and "too fast" loops may lead to overly rapid decisions. In the following sections, we adopt the rule system of Appel (see Sect. 5.1) to implement our rule-based manager programs.

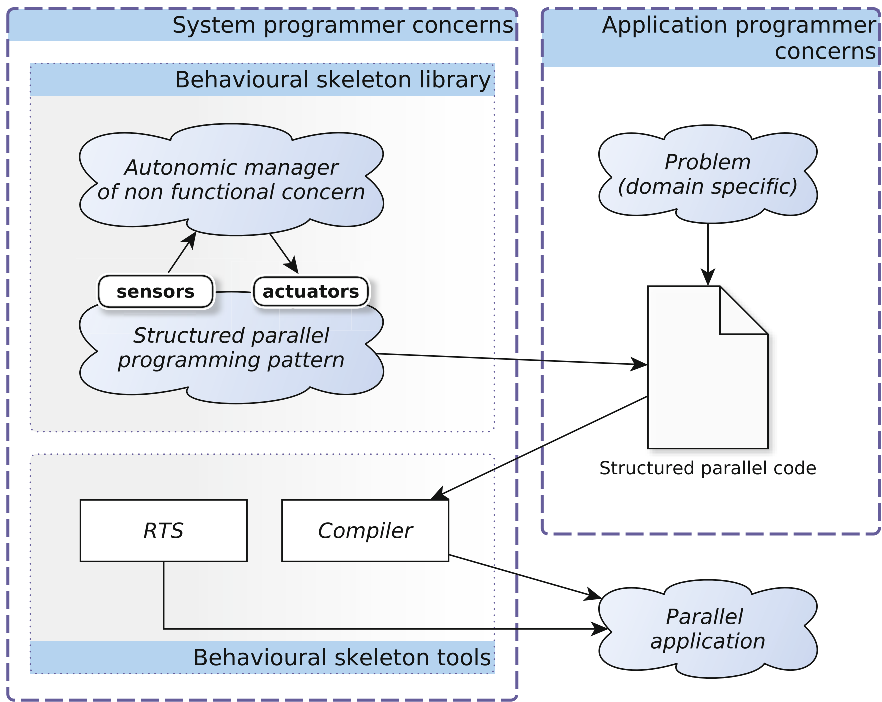

Fig. 3. Behavioural skeletons

Appel chooses rules for firing using a loop such as that mentioned above. However, the trigger events are gathered continually and an ordered list of events is exposed to the rule system at each loop iteration.

It's worth pointing out here that "planning" activities in our MAPE loop are not actually *proper* planning activities. Rather, the "plan" step in the loop consists in applying a plan that has been already coded in the action part of the ECA rules used as the program of the autonomic manager. These rules may also include an action part that somehow modifies the rule set. For example, a rule priority may be lowered or a rule may be substituted by a different one which more precisely reflects the actions needed in the current situation. This notwithstanding, the "plan" phase is actually a kind of "actuate one of the already established plans" phase. At a rather higher level of abstraction, the process leading to the design of the rules used as the program of the MAPE loop is a kind of MAPE loop itself. The current situation is monitored and then it is analyzed. During the analysis phase a policy is eventually identified which turns into a plan to be actuated/executed by generating suitable loops for our run time MAPE loop.

3.1 Behavioural Skeletons

Building on the concepts detailed in the previous Sections, we proposed some time ago the concept of *Behavioural skeleton*, i.e. of a parallel design pattern coupled with an autonomic manager taking care of a non-functional concern. In the original behavioural skeleton design, the parallel design patterns considered were the traditional ones in stream parallel computing models, that is *task farm* and *pipeline*. Task farm (a.k.a. abstraction of the master/worker implementation pattern) completely captures and models embarrassingly parallel computation on streams. Pipeline, instead, captures and models computation in stages, without backward communications. Also, the original behavioural skeleton design considered management of only a single non-functional concern: performance.

The behavioural skeleton approach is outlined in Fig. 3. A behavioural skeleton library is made available to the application programmer. The library contains several composable behavioural skeletons. Each behavioural skeleton consists of a parallel design pattern *and* of an autonomic manager running a MAPE loop and using an ECA rule system to implement policies. Suitable sensors and actuators are implemented within the parallel design pattern implementation to support autonomic manager activities.

The application programmer in charge of writing a parallel application chooses a behavioural skeleton or some composition of behavioural skeletons from the BS library and provides the behavioural skeleton(s) business logic parameters. For example, if the pattern used to express parallelism within the application is a pipeline, the application programmer chooses the pipeline behavioural skeleton and instantiates it passing as parameters the code (wrappers) implementing the business logic of the pipeline stages. If one of the stages has to be further parallelized, the application programmer may pass as pipeline stage an instance of a task farm whose worker parameter implements the parallel stage business logic.

Once the application programmer has written his/her application using behavioural skeletons, a compiler takes care of producing suitable parallel code for the target architecture at hand. This code relies on the functions provided by the behavioural skeleton run time library, of course.

It is worth pointing out several notable features of this approach:

- the system concerns (those requiring specific knowledge concerning the target architecture/system at hand) and the application concerns (those requiring more domain specific knowledge related to the application field) are kept completely separate. Separation of concerns clearly enforces productivity and efficiency in both application and system programmer activities.
- the application time-to-deploy is significantly reduced by reuse of behavioural skeleton library components.
- performance portability across different architectures is the responsibility of system programmers (as opposed to application programmers) who provide, in the behavioural skeleton library, components specific for the different target architectures.

– policy programmability is ensured by the ECA rule system embedded in the autonomic manager MAPE loop. Programming rules (declarative style) is much more user friendly and efficient than writing specific code using the sensors and actuators provided by the associated design pattern.

Leveraging on all these attractive properties, a prototype implementation of behavioural skeletons on top of the ProActive/GCM middleware has been demonstrated to be able to carefully manage performance in stream parallel applications [3].

4 Conflict Detection and Resolution in Rule-Based Systems

Policy conflict has been recognized as a problem and there have been some attempts to address it, mostly in the domain of access or resource control [17].

Kind of Conflicts. ECA rules conflict if (1) they may be triggered at the same time and (2) their conditions overlap and (3) their actions conflict. While this definition makes complete sense only in a specific application domain, as one must be aware of what conflicting actions are, the problem is inherent to policies. To ensure that policies can be applied it is necessary to remove conflicts. This process involves two stages: first one needs to identify whether conflicts can occur, that is to *detect* conflicts, and then to remove them, that is *resolve* conflicts.

The general definition of policy conflict can be extended to accommodate some special cases.

In [8] the authors discuss what it means for two rules to be triggered at the same time, providing two different interpretations: their trigger sets overlap (and the actual triggering event is in the overlap) or the action of one is in the trigger set of the other. The former case has been called STI (Shared Trigger Interaction) and the latter SAI (Sequential Action Interaction).

More generally, and this provides interesting future work in our case, the designer may be interested in specifying conflicts on the basis of traces, i.e. define as conflicting rules that (1) may be applied within n MAPE loops and (2) whose actions conflict.

One further aspect to consider, and this is again based on experience in feature interaction, is the question as to how many policies are required to generate a conflict. In the community, discussions have taken place around a topic called “three-way interaction”. In the feature interaction detection contest at FIW2000[9] this was an issue, and the community decided that there are two types of three-way interaction: those where there is already an interaction between one or more pairs of the three features and those where the interaction only exists if the triple is present. The latter were termed “true” three-way interactions.

Nothing has been written about true three-way interaction, as only one, quite contrived, example of such an interaction has been found. Hence we consider realistic to assume that no "true" three-way interaction may occur, and limit ourselves to pair wise checking.

Conflict Detection Time. We distinguish between design time (static) and run-time detection. In run-time detection, conflicts, if any, are looked for at each execution step among the rules that can be applied at the step. When conflict detection is anticipated at design-time, rules are filtered before being entered in the policy engine, to detect those that would originate conflicts. In this way we can provide the user with confidence that the rules are conflict free.

In our former works we addressed design time detection. In [13,14], we take a logic–based approach to this end: conflicts are detected by deducing specific formulae in a suitable temporal logic theory. In [6] we exploit the use of model checking to detect policy conflict. This is the approach employed in this paper.

Layouni et al. in [11] also experimented with the use of the model checker Alloy [5] to support policy conflict resolution.

Conflict Resolution. Conflict resolution can in general be attempted in a number of ways, and which is best suited depends on the situation. We can broadly distinguish between resolution at design–time and resolution at run–time. The taxonomy of policy conflict in [17] makes explicit that design-time resolution is always feasible when policies are co-located and owned by the same user. In this case resolution will be a redesign of the policies. However, when policies are distributed, this is not always possible and it is preferable to deal with conflicts at run-time. Resolution in this case may exploit priorities among policies, activating only policies with greater precedence. Nevertheless, there is a wide spectrum between the two extremes of co-location and complete distributed placement, and any conflict that is resolved before run-time is of benefit.

A comprehensive survey on detection and resolution techniques in three well-known policy management approaches, KAoS, Rei and Ponder, is found in [20].

5 Multiple Non-functional Concern Management: Formal Tool Support

The ability to develop independent managers and to modify them to accomplish coordinated management of multiple concerns is attractive for two reasons: it enforces modular design and reuse; and allows better use of domain specific knowledge of different non-functional concerns.

However, combining a set of single-concern managers may be difficult to achieve since it requires expertise in *all* of the non-functional concerns to be coordinated, and because the sheer number of evolution paths of the combined managers may make it extremely difficult for the human to identify the possibility of a conflict arising.

Model checking tools may provide fundamental support, however, as proposed in [6,10]: "conflicts" can be *detected* by model checking, once the conflicting atomic actions have been identified. More precisely, the whole design phase includes the following steps:

- Independent experts design and implement policies relative to distinct non-functional concerns.
- A set of conflicting actions is defined, such that a pair of actions a_i, a_j are in the set *iff* action a_i "undoes" action a_j and vice versa.
- A formal model of the rule system is derived, which is fed to a model checker.
- The model checker is used to check formulas stating that conflicting actions may occur "at the same time", that is in the same MAPE loop iteration.
- The traces leading to the situation with the conflicting actions obtained from the model checker are used to change the rules to handle conflicts.[1]

5.1 An Experiment in Static Conflict Detection for Autonomic Managers

In the sequel, we first describe our experimental setting, and then discuss our first results, with respect to the likelihood of applying the technique to real life examples. The ingredients of the technique are a policy language, a model checker and a translator able to generate a checkable model from the policies.

Appel. We use Appel [22,21] to write the management rules. Appel is a general language for expressing policies in a variety of application domains: it is conceived with a clear separation between the *core* language and its specialization for concrete *domains*, a separation which turns out very useful for our purposes.

In Appel a *policy* consists of a number of *policy rules*, grouped using a number of operators (**seq**uential, **par**allel, **g**uarded and **u**nguarded choice).

A policy rule has the following syntax:

[**when** *trigger*] [**if** *condition*] **do** *action*

The core language defines the structure but not the details of these parts, which are specifically defined for each application domain: base triggers and actions are domain-specific atoms; an atomic condition is either a domain-specific or a more generic (e.g. time) predicate. This allows the core language to be used for different purposes. In our case, as mentioned above, the triggers relate to active sensors, conditions to (passive) sensors, and actions to actuators.

Triggers can be combined with a disjunction, complex conditions can be built with Boolean operators, and a few operators (**and**, **andthen**, **or** and **orelse**) are available to create composite actions.

[1] At the moment conflicts are identified by the model checker, but then the actions needed to resolve the situation (i.e. the modifications to the manager rules) are performed by humans. The asymptote is to have this part also executed automatically.

The semantics of Appel [,] which before was only defined informally, as with most policy languages, has been formally defined by translation into the temporal logic ΔDSTL(x) [15,16].

Though Appel supports also a notion of priority among the rules, we do not exploit this currently.

UMC. This is an on-the-fly analysis framework [12,23,19] that allows the user

1. to interactively explore the behaviour of a UML state machine;
2. to visualize abstract slices of its behaviour; and
3. to perform local model checking of UCTL formulae, UCTL being a branching-time temporal logic [7].

The last feature is the most important for our purposes, but the previous ones are very useful once a conflict is detected and we need a deep understanding of what is happening to resolve it.

UCTL allows specification of the properties that a state should satisfy and combination of these basic predicates with advanced temporal operators dealing also with the performed actions. Some care must be taken in writing the formulae that characterize the conflicts to be detected, since they are checked not against the traces of the UML state machine, but against the traces of an equivalent standard state machine – generated by UMC – where parallelism is resolved with interleaving. So to detect two conflicting parallel actions one has to detect any sequence of the actions in any path in the traces.

Appel2UMC. We have defined a semantics-preserving compositional mapping from Appel to UML, suitable for model checking with UMC. Since UMC operates on UML state machines, the target of the mapping happens to be a subset of UML state machines: policies and policy groups are defined using composite states, i.e. states with structure reflecting the one imposed by the Appel operators onto policies and actions.

To derive a UML state machine model of the system to feed the checker, we follow the approach of [6]: Appel policies are automatically mapped to a UMC specification, i.e. the description of a UML state machine, in the UMC textual input format.

The mapping is based on the Appel semantics given in terms of UML state machines. Actually, the mapping needs not consider the actual semantics of the actions, but only an abstract one, where an action may result in a *success* or a *failure*. Intuitively, these notions entail that an action may complete normally (success) or may abort for some reason (failure), and Appel leaves the specifics of an action success or failure to the domain. However, it defines the success or failure of a composed action as a composition of the successes and failures of the actions under composition. Therefore, for the translation, actions can be treated as propositional atoms.

The prototype translator from Appel to an equivalent UMC specification, dubbed *Appel2UMC*, is written in OCaml, and structured in a syntax definition

module, a *Compiler*, and an *Unparser*. *Compiler* translates APPEL to UMC, at the abstract syntax level, and *Unparser* generates the textual version needed by the model checker. These core modules depend on a further one that defines the domain dependent features (triggers, conditions and actions), thus ensuring adaptability of the tool. At the moment, the syntax is about 100 lines, the core modules are slightly over 500 lines, and the domain dependent part less than 80 lines. Translation times are not an issue.

5.2 Preliminary Results

To evaluate the feasibility of the approach, we ran some experiments using the model checker UMC [23,19] to verify part of the policies introduced in [4] for structured parallel computations.

We consider two independently developed managers controlling respectively performance and power consumption of an application with a *farm* structure.

Managing Performance. The APPEL rules in Table 1 address *performance* management: the first two capture a noteworthy change in performance (trigger *NewPerformanceMonitored* and the others a noteworthy change in the parallelism degree of the execution. What a *noteworthy change* is, is defined by the semantics of the active sensors that generate the events matching these triggers.

These changes may be disregarded if they do not take the system outside of the "normal" operational range, i.e., when neither *LowPerformance* nor *HighPerformance* (*LowParDegree* nor *HighParDegree*, respectively) holds, that is, when the values returned by the corresponding sensors do not satisfy the intended condition.

Let us now consider what happens when PM1 fires, i.e., when performance drops below the threshold. The goal obviously being to reestablish an acceptable level, a new worker is introduced, in two macro steps, each sequencing two basic actions on the current state of the application. In the best of worlds, an available processor is allocated to the farm (*GetResource*), the appropriate run-time-support is deployed (*DeployRts*)and started (*StartRts*), and finally the new worker is linked (*LinkRts*)and therefore made available to the farm.

What if something goes wrong in the execute phase, e.g. no more processors are available? PM1 is written (like all the other rules in this simplistic scenario, by the way) using the composition operator **andthen** in such a way that the failure of any basic action entails the failure of the rule as a whole, and therefore the rule fires but has no effect whatsoever.[2]

The other rules were designed similarly, and use a few more basic actions, whose meaning should be immediate. Only *GetWorker* may need a comment: it selects one of the active workers in the farm, likely so that the manager can consequently free the resources it is using.

[2] Actually, care must be taken that the controlled application is rolled-back to its initial state. Also, in a realistic scenario, some alarm should be sent to the administrator, when appropriate.

Table 1. The Performance Manager Rules

PM1: **when** *NewPerformanceMonitored*
if *LowPerformance*
do (*GetResource* **andthen** *DeployRts*) **andthen**
(*StartRts* **andthen** *LinkWorker*)
PM2: **when** *NewPerformanceMonitored*
if *HighPerformance*
do (*GetWorker* **andthen** *UnlinkWorker*) **andthen**
(*StopRts* **andthen** *UndeployRts*)
PM3: **when** *NewParMonitored*
if *LowParDegree*
do (*GetResource* **andthen** *DeployRts*) **andthen**
(*StartRts* **andthen** *LinkWorker*)
PM4: **when** *NewParMonitored*
if *HighParDegree*
do (*GetWorker* **andthen** *UnlinkWorker*) **andthen**
(*StopRts* **andthen** *UndeployRts*)

Managing Power Consumption. The APPEL rules in Table 2 address this concern. They should be easily understandable, at this point, since they use many of the actions already used for performance management, but react to different events and are subject to new appropriate conditions.

The two rules deal only, in different ways, with the need to decrease power consumption. PCM1 takes a drastic approach, and kills one of the workers, to get the result. PCM2 attempts to save something, trading away one of the more power consuming workers for a less consuming one.

Conflict Definition. As we have seen, both managers operate on the application graph by executing actions like *LinkWorker* and *UnlinkWorker*, which include or remove a node in/from the current computation, respectively.

These actions are marked as an “atomic conflict”, as they nullify each other, if performed in the same control cycle.

Putting UMC to Work. To illustrate how conflict detection is supported by UMC, we consider as simple a situation as possible, with only two rules, one from each manager, namely PM1 and PCM1. Given the parallel composition of these two rules, Appel2UMC generates the textual representation of the corresponding UMC model, dubbed *System* by default.

Loading *System* into the framework, an equivalent graphical representation (Fig. 4) is generated by the framework: it is a translation of the input *System* model into a standard automaton, resolving parallelism with interleaving: in this representation, “parallel” actions in the rules appear in sequence, in all possible different orders, along several traces. It is precisely the space of traces of this automaton that is searched by the UMC model checker.

Table 2. The Power Consumption Manager Rules

PCM1:	**when** *NewPowerConsumptionMonitored* **if** *PowerContractLow* **do** (*GetWorker* **andthen** *UnlinkWorker*) **andthen** (*StopRts* **andthen** *UndeployRts*)
PCM2:	**when** *NewPowerConsumptionMonitored* **if** *PowerContractLow* **do** [(*GetPowerWorker* **andthen** *GetCheaperWorker*) **andthen** (*UnlinkWorker* **andthen** *StopRts*)] **andthen** [(*UndeployRts* **andthen** *DeployRts*) **andthen** (*StartRts* **andthen** *LinkWorker*)]

In this simple example it is clear, by inspection of the automaton, that the conflict will arise. However, as the number of rules in parallel increases, the size of the space of the traces of the corresponding automaton increases exponentially, and human inspection becomes quickly infeasible.

To use model checking instead, we need first to formalize the relevant question *may a conflict occur in one MAPE cycle?* in UCTL, in terms of traces: is there no trace among those generated by the automaton, which includes both *LinkWorker* and *UnlinkWorker*? Formally, the question is expressed by requiring that it should never be the case that there is a path were a *LinkWorker* (*UnlinkWorker*) state has a path to a subsequent *UnlinkWorker* (*LinkWorker*), that is:

$$\begin{gathered}(\mathsf{not\ EF\ EX}\{\mathsf{LinkWorker}\}\ \mathsf{EF}\{\mathsf{UnlinkWorker}\}\ \mathsf{true}) \\ \& \\ (\mathsf{not\ EF\ EX}\{\mathsf{UnlinkWorker}\}\ \mathsf{EF}\{\mathsf{LinkWorker}\}\ \mathsf{true})\end{gathered} \tag{1}$$

Our aim is to specify that there is no single MAPE cycle where both actions *LinkWorker* and *UnlinkWorker* are executed. The question has to be formulated in this way, since UMC translates the input model into a standard finite state machine, resolving parallelism with interleaving: "parallel" actions appear in sequence, in different orders, in several traces.

Running the model checker, it gives "false" as answer, and the explanation of this result gives the traces leading to the situation where the formula is demonstrated false.

To conclude we remark that the logical formulae associated with conflicts can be systematically written by the designer following the pattern of formula (1).

Resolving the Conflict. According to the method outlined in Sec. 5.1 we should be able to collect all the knowledge necessary to produce a modified

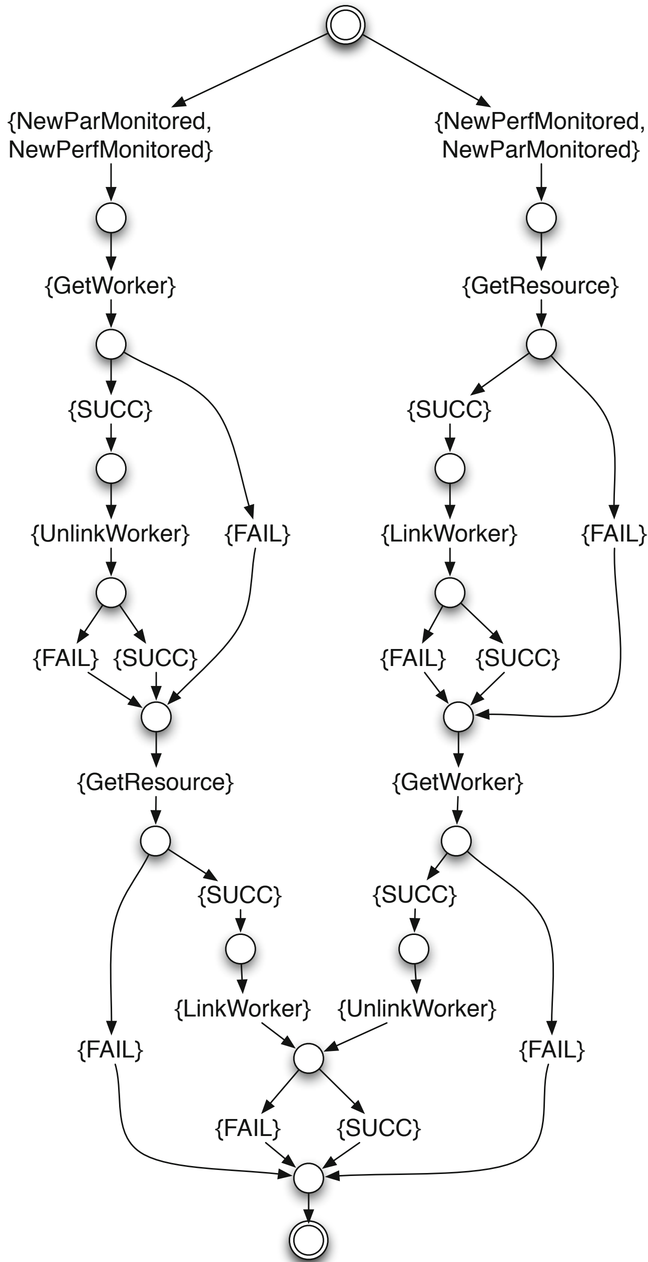

Fig. 4. Representation of *System* as an automaton

set of rules properly handling the conflict from the answer given by the model checker to explain why the model falsify formula (1):

- the situation leading to the conflict is determined by the contemporary firing of the power manager rule PCM1 to "reduce power usage", and of the performance manager rule PM1 to "increase parallelism degree".
- there is at least one path leading to the conflict, which includes the actions in PM1 and PCM1.

Based on this knowledge, we can conclude that handling of the detected conflict may be achieved by a high priority rule (or a set of rules):

- whose (new) trigger logically corresponds to the conjunction of the two triggers as APPEL, and most ECA based notation, does not support the conjunction of triggers but only trigger disjunction, and
- whose action part consists in a plan whose effect is an increase of the parallelism degree with reduced power consumption.

Alternatively, we may solve the conflict by assigning a priority to one of the conflicting rules, in such a way that only the highest priority rule is executed.

Feasibility. We discussed a very simple example: two rules giving rise to a very compact model and useful "explanations" in terms of traces. The number of states generated in the UMC model is below one hundred and the response time of the model checker is of the order of a fraction of a second.

We made a few slightly more realistic experiments using up to all the rules given above. The times needed to execute the model checker with different rules sets and queries are in the tens of milliseconds range: when 2, 4 or 6 rule systems are used, the time to model check the "conflict exists" formula are 30, 50 and 60 msecs, respectively (the model checker was running on a quad core Core Duo Intel Xeon workstation). When the AG(true) is model checked–this query gives the upper bound in execution times, as it requires the model checker to visit all possible paths in the model–the time spent in the model checker is 20, 120 and 250 milliseconds, respectively. These results seem to confirm that the approach is feasible in more realistic situations. We cannot show the involved automata, as the graphs are significantly larger and do not fit easily on a page.

6 Conclusions

We discussed formal tool support for the integration of independently developed autonomic managers, each taking care of a different non-functional concern. The formal tool support provides suitable hints to the programmer integrating these independently developed managers into a single parallel applications. As the manager programs are suitable sets of ECA rules, the formal tool support provides evidence of the conflicting rules in different managers as well as of the

initial situations (states) that eventually lead to the conflicting actions generated by the ECA rules. The preliminary results demonstrate the feasibility of the approach and the relatively modest computational cost of the model checker activities involved.

Paraphrase Perspective

The research results discussed in this paper will be exploited within the ParaPhrase project in various ways.

First, although not discussed here for the sake of simplicity, the ECA rule sets we are considering in our non-functional concern managers include rules that change the structure of the parallel computation. For example, a manager taking care of performance in a program whose parallel structure may be represented as a *pipeline(seq(f), seq(g), seq(h))* may discover that the second stage takes much longer to execute than the first and the third. Therefore he may execute an action aimed at transforming the program into a *pipeline(seq(f), farm(seq(g)), seq(h))*. As the main focus of ParaPhrase is on parallel program refactoring, these rules transforming parallel pattern compositions to better performing parallel pattern compositions represent natural candidates for use in the refactoring process.

Second, we foresee the possibility to implement some kind of dynamic management of the re-factoring to suit the varying conditions on the target architectures within ParaPhrase. The techniques discussed here will naturally suit the need to verify that no conflicts are generated while dynamically re-factoring our parallel applications.

References

1. Aldinucci, M., André, F., Buisson, J., Campa, S., Coppola, M., Danelutto, M., Zoccolo, C.: Parallel program/component adaptivity management. In: Gorlatch, S., Danelutto, M. (eds.) Proc. of the Integrated Research in Grid Computing Workshop, Pisa, Italy, TR-05-22, pp. 95–104. Università di Pisa, Dipartimento di Informatica (2005)
2. Aldinucci, M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D., Tonellotto, N.: Behavioural skeletons for component autonomic management on grids. In: CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion, Crete, Greece (June 2007)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: Proc. of Intl. Parallel & Distributed Processing Symposium, IPDPS, Rome, Italy, pp. 1–12. IEEE (May 2009)
4. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic managenemt of multiple non-functional concerns in behavioural skeletons. In: Proc. of the CoreGRID Symposium 2009, CoreGRID, Delft, The Netherlands. Springer (August 2009)
5. Alloy Community, `http://alloy.mit.edu/community/`
6. ter Beek, M., Gnesi, S., Montangero, C., Semini, L.: Detecting policy conflicts by model checking UML state machines. In: Reiff-Marganiec, S., Nakamura, M. (eds.) Feature Interactions in Software and Communication System X, pp. 59–74. IOS Press (2009)

7. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008)
8. Calder, M., Kolberg, M., Magill, E.H., Marples, D., Reiff-Marganiec, S.: Hybrid solutions to the feature interaction problem. In: Amyot, D., Logrippo, L. (eds.) FIW, pp. 295–312. IOS Press (2003)
9. Calder, M., Magill, E.H.: Feature Interactions in Telecommunications and Software Systems VI, Glasgow, Scotland, UK, May 17-19. IOS Press (2000)
10. Danelutto, M., Kilpatrick, P., Montangero, C., Semini, L.: Model Checking Support for Conflict Resolution in Multiple Non-functional Concern Management. In: Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Di Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S.L., Traff, J.L., Vallée, G., Weidendorfer, J. (eds.) Euro-Par 2011 Workshops, Part I. LNCS, vol. 7155, pp. 128–138. Springer, Heidelberg (2012)
11. Layouni, A.F., Logrippo, L., Turner, K.J.: Conflict detection in call control using first-order logic model checking. In: du Bousquet, L., Richier, J.-L. (eds.) Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems, France, pp. 77–92. IMAG Laboratory, University of Grenoble (2007)
12. Mazzanti, F.: UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR (2006)
13. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic–Based Detection of Conflicts in Appel Policies. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 257–271. Springer, Heidelberg (2007)
14. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamenta Informaticae 89(4), 511–538 (2008)
15. Montangero, C., Semini, L.: Distributed states logic. In: 9th International Symposium on Temporal Representation and Reasoning, TIME 2002, Manchester, UK. IEEE CS Press (July 2002)
16. Montangero, C., Semini, L., Semprini, S.: Logic Based Coordination for Event–Driven Self–Healing Distributed Systems. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 248–262. Springer, Heidelberg (2004)
17. Reiff-Marganiec, S., Turner, K.J.: Feature interaction in policies. Comput. Networks 45(5), 569–584 (2004)
18. Ruz, C.: Autonomic Monitoring and Management of Component-Based Services, PhD Thesis. Univ. de Nice - Sophia Antipolis (2011)
19. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. 76(2), 119–135 (2011)
20. Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N., Uszok, A.: Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 419–437. Springer, Heidelberg (2003)
21. Turner, K.J., Reiff-Marganiec, S., Blair, L., Pang, J., Gray, T., Perry, P., Ireland, J.: Policy support for call control. Computer Standards and Interfaces 28(6), 635–649 (2006)
22. Turner, K.J., Reiff-Marganiec, S., Blair, L., Cambpell, G.A., Wang, F.: Appel: An adaptable and programmable policy environment and language. Technical Report TR-161, University of Stirling (December 2009)
23. UMC v3.5, `http://fmt.isti.cnr.it/umc`