Model Checking Support for Conflict Resolution in Multiple Non-functional Concern Management

Marco Danelutto[1], P. Kilpatrick[2], C. Montangero[1], and L. Semini[1]

[1] Dept. Computer Science, University of Pisa
[2] Dept. Computer Science, Queen's University Belfast

Abstract. When implementing autonomic management of multiple non-functional concerns a trade-off must be found between the ability to develop independently management of the individual concerns (following the separation of concerns principle) and the detection and resolution of conflicts that may arise when combining the independently developed management code. Here we discuss strategies to establish this trade-off and introduce a model checking based methodology aimed at simplifying the discovery and handling of conflicts arising from deployment–within the same parallel application–of independently developed management policies. Preliminary results are shown demonstrating the feasibility of the approach.

Keywords: Autonomic managers, model checking, non-functional concerns, structured parallel computations.

1 Introduction

The past ten years have seen a major shift in the nature of distributed and parallel computing systems. While traditionally systems were relatively unchanged throughout their lifetime and existed in more or less stable environments, the pervasive nature of many modern systems and their composition from grid or cloud services mean that often they need the capability to adapt automatically to changes in their environment and/or changes to their constituent services. This has relatively little impact on the core functionality, which tends to lend itself to precise definition, but has significant implications for non-functional aspects such as performance, security, etc. These aspects may not be so clearly defined and often the code to handle them is interwoven with the core functionality. In previous work we have proposed a means of isolating such code by introducing, in the notion of behavioural skeleton, an amalgam of algorithmic skeleton (parallel pattern) – for example, farm, pipeline – together with one or more managers of non-functional concerns (such as performance, power usage, security). A manager monitors the performance of its associated skeleton with respect to a particular concern and has the capacity to initiate changes to the skeleton behaviour with respect to that concern. However, when bringing together managers of differing concerns, these managers may not sit comfortably together and indeed may be in conflict. For example, it is conceivable that a power manager may be prompting removal of a worker from a task farm (to reduce power consumption) while a performance manager is (more or less) simultaneously indicating that a worker be added to boost performance. In earlier work [2] we proposed a protocol for coordinating the activities of managers to deal with such conflicts and showed how this protocol might operate in practice [3]. There the conflicts were identified simply by human inspection. Here we extend that work by using a model checking approach to identify potential conflicts and generate a trace showing the origin of the conflict. This trace allows us to modify the design to avoid the possibility of conflict or to deal with such conflict dynamically.

2 Rule Based Management

In [1] we introduced the concept of a behavioural skeleton comprising an algorithmic skeleton (parallel pattern), e.g., farm, pipe, etc. *and* an autonomic manager taking care of non-functional aspects of the computation, such as performance, security, etc. The manager has the capacity to modify the behaviour of the associated skeleton by prompting a change to the structure or operation of the skeleton. It runs a classic MAPE loop: it *M*onitors the behaviour of the pattern with respect to a given non-functional concern (e.g., throughput for performance); *A*nalyses the monitored values to determine if the skeleton is operating satisfactorily; if adjustment is necessary it *P*lans for a modification; and finally *E*xecutes the modification. The MAPE loop then recurs. The MAPE cycle may in practice be implemented as a set of *pre-condition→action* rules [3].

From a separation of concerns viewpoint it is best if autonomic management can be developed as a set of separate managers, each handling a single concern. Each manager may thus be developed by an expert in the concern at hand. The challenge then lies in coordinating the activities of these managers so that cooperation rather than conflict within a given MAPE cycle is achieved. To identify conflicts it is necessary to have a means of cataloguing the structural or operational changes to a skeleton that may be initiated by a manager. For this it is useful to have the concept of an *application graph* [2] whose nodes represent parallel/distributed activities and whose arcs represent communications/synchronizations among these activities (Fig. 1). Each node and arc can be labelled with metadata specifying non-functional properties. To identify conflicts one must first identify actions on the application graph which are clearly in opposition (such as add-worker/remove-worker) and determine if there is a possible evolution of the system which would lead to two such actions being invoked in the same MAPE cycle. It is in this latter activity that model checking proves beneficial, as will be seen in section 4. First we discuss more generally strategies for managing non-functional concerns in distributed systems.

3 Multiple Non-functional Concern Management

When dealing with multiple non-functional concerns within a parallel application, conflicts may arise when two or more management policies demand changes

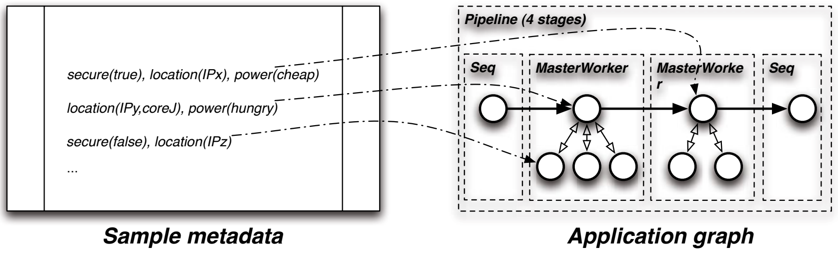
secure(true), location(IPx), power(cheap)
location(IPy,coreJ), power(hungry)
secure(false), location(IPz)
...
Sample metadata
Pipeline (4 stages)
Seq
MasterWorker
MasterWorke r
Seq
Application graph

Fig. 1. Sample Application Graph

to the application graph which are incompatible, as discussed above. Here we will describe three approaches to resolution of such conflicts. These vary in the degree of coordination required and in the timing of this coordination. In principle, autonomic non-functional concern management of parallel or distributed computations should be designed by experts in parallel computing *and* in the single non-functional concern at hand to ensure the maximum impact of the management. The approaches presented differ in how they attempt to resolve the tension that exists between separation of concerns at the development stage and the need to bring these concerns together in a unified running system.

3.1 Fully Coordinated Co-design

In the *fully coordinated design* approach we sacrifice separation of concerns for ease of consolidation. A single expert (or a single team of experts) is in charge of developing management policies for all the non-functional concerns considered. As a result, the management policies may be coordinated from the beginning and conflict may be *prevented by design*.

Situations raising conflicts are detected by the single (team of) expert(s) and rules in the managers are (re)programmed so that their combined effect no longer raises conflicts. For example, consider the case where performance security and power consumption are the non-functional concerns to be managed. When programming rules to increase the program throughput by augmenting the parallelism degree, both recruitment of non power-hungry processing resources and the deployment of secure communication mechanisms will be considered. As a result of consideration of both concerns, a simple rule stating that if throughput is low[1] the parallelism degree should be increased:

R1 performance low $\rightarrow$ recruit resource; deploy code; link resource to par. computation;

will be replaced by the following pair of rules:

R1a performance low **and** R is available and low power resource **and** secure(R) $\rightarrow$ recruit(R); deploy code; link R to parallel computation;
R1b performance low **and** R is available and low power resource **and** unsecure(R) $\rightarrow$ recruit(R); deploy secure code; link R to parallel computation;

merging knowledge of performance, security and power management concerns.

[1] e.g. with respect to some user agreed SLA or *contract*.

3.2 Coordinated Decision Commitment

In the *coordinated decision commitment* approach, emphasis is placed on separation of concerns and an attempt is made to build into the rules for each separate concern the ability to identify and resolve conflicts *at runtime*. Here distinct (teams of) experts design the policies for the autonomic management of distinct non-functional concerns. However, the policies are designed in such a way that they can also work when policies for the management of other concerns are concurrently applied in the same context. In a sense, a coordinated commitment of independently taken decisions is implemented – hence the name.

To achieve this coordination a two-phase distributed agreement protocol such as that proposed in [2] may be adopted. In this case, when according to the policies managing concern C_i a decision d_j is to be taken, a *consensus* is sought from managers of all other concerns different from C_i. The consensus is sought on the new application graph resulting from the implementation of decision d_j. If all the other managers agree on the feasibility of the new graph, the decision is taken. If at least one of the other managers (e.g. the one managing concern C_k) indicates that the decision would eventually lead to an unfeasible application graph – according to the concern managed, C_k – then the decision is aborted and the priority of the rules firing the decision is lowered. The last, and more interesting, case is where all managers agree on the feasibility of the new application graph, but some request that an additional feature be taken into account when implementing the decision d_j. In this case the decision is committed by the manager managing concern C_i using an alternative implementation plan which ensures the additional requirements.

If we consider the rule R1 discussed in Sec. 3.1 leading to an increase of the current parallelism degree, in this case we will eventually arrive at the set of rules:

R1.1 performance low **and** R is available $\rightarrow$ ask consensus on recruitment of resource R to other managers
R1.2 all managers grant unconditional consensus $\rightarrow$ recruit resource R; deploy code; link resource to parallel computation
R1.3 one manager negates consensus $\rightarrow$ abort decision; lower priority for R1.1
R1.4 all managers grant consensus provided properties $P_1, \ldots, P_k$ are ensured $\rightarrow$ change original decision plan A' to A'' such that A'' ensures P_1 to P_k; commit decision through plan A''

In this case, the knowledge needed to implement a different decision plan comes in part from the knowledge of the concern of the manager using these rules and in part from the concerns managed by autonomic managers requiring properties P_i. If consensus is granted, provided that P_0 = **security** is ensured, then the original plan:

recruit(R); deploy code; link R to parallel computation;

will be substituted by the new plan:

recruit(R); deploy secure code; link R to parallel computation;

With this approach the separation of concerns principle is compromised somewhat by the need for the individual rule systems to be designed so as to be able to accommodate future interactions with rule systems pertaining to other concerns. Insofar as the interactions are described in terms of modifications to the application graph, this can reasonably be achieved, although the fact that the graph contains metadata relating to various concerns (and not just structure) means that full separation is not possible.

3.3 *Ex Post* Independent Manager Coordination

Coordinated commitment may be regarded as an *interpretive* approach to coordination. When we deploy more than a single manager in the same application, we pre-configure a number of conflict situations that may eventually arise (managers ask consensus) and pre-configure modified decision commitment plans taking care of these conflicts.

The consensus building phase takes time, however. Even in the case of no conflicts, communications must be performed between the manager taking the decision and those checking the decision is safe w.r.t. their own policies. This means of implementing management is thus a *reactive* process, and the reaction time of the system critically impacts the efficiency of the whole management process. Thus any delay should be reduced as much as possible.

As is usual in computer science, moving from interpreters to compilers improves performance.

In the third approach, we retain the idea of coordinated decision making but implement it through the *compilation* of a modified set of management rules *before* the system is actually run. In particular, we perform the following steps:

- We analyze the independently developed sets of rules, looking for those rules which, if fireable at the same time, may lead to conflicting actions. The key point here is again to identify the conflicting actions in terms of the application graph.
- Then we derive a new set of rules possibly including some (modified version) of the initial set of rules from the different managers together with new rules managing the conflict situations.

Note that the knowledge required to determine this modified set of rules is roughly the same as that needed to implement the coordinated decision commitment approach.

For example, consider a performance manager having a rule that increases the parallelism degree when low throughput is experienced ($R_{pd-increase}$), and a power manager with a rule stating that "power hungry" resources must be dismissed when too much power is consumed by the computation ($R_{pw-decrease}$). In this case a conflict arises if the performance manager wants to increase the parallelism degree and the power manager wants to dismiss a resource. Adopting an "ex post coordination" approach, we can look at the condition of the rules used in the conflict situation and implement a new rule, with a higher priority w.r.t. to both $R_{pd-increase}$ and $R_{pw-decrease}$, stating that if performance is low and power consumption high we should increase parallelism degree by selecting low consumption resources or by releasing high consumption resources and replacing them with a larger number of low consumption resources.

4 Model Checking for Conflict Resolution

The ability to develop independent managers and modify them to accomplish coordinated management of multiple concerns looks attractive in that it enforces modular design and reuse as well as allowing better use of domain specific knowledge relative to different non-functional concerns.

However, combining a set of single-concern managers in both coordinated decision commitment and ex post independent manager coordination may be difficult to achieve unless the developer is an expert in *all* of the non-functional concerns to be coordinated. Even then, the sheer number of evolution paths of the combined managers may make it extremely difficult for the human to identify possible conflict.

Model checking tools may, however, provide useful support. When considering a complex set of management rules, such as those describing a number of different non-functional concern managers, an approach such as that proposed in [9] can be used. There, "conflicts" in rule based management systems can be *detected* using a model checker after identifying conflicting atomic actions.

Here we modify that methodology to support conflict detection in rules describing independently developed managers. As the "ex post" approach looks the more promising, we consider the use of a model checker to support compilation of a coordinated set of rules from a set of independently developed rules. We propose a methodology in which:

- Independent experts design policies for distinct non-functional concerns. We assume the rules are expressed using APPEL [10]. This allows better structuring of the manager rules. In particular, we use APPEL triggers 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. By using the concept of APPEL triggers to start rule evaluation, we avoid problems related to MAPE loop polling.
- 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. As actual atomic actions only affect the application graph, this step does not require any specific knowledge of non-functional concerns.
- A formal model of the system is derived, which is fed to a model checker. The model is generated following the approach outlined in [9]. APPEL policies are automatically mapped to a UMC specification, i.e. the textual description of a UML state machine, in the UMC input format. The mapping is based on the APPEL formal semantics, as given in [7].

- The model checker is used to check formulas stating that conflicting actions may coincide, that is occur in the same MAPE loop iteration. Traces of actions leading to conflicts are produced.
- Knowledge obtained from the traces is used to develop the additional rules to be included in the rule system to handle conflicts.[2]

5 Preliminary Results

To evaluate the feasibility of the proposed approach, we ran experiments using the model checker UMC [11, 8]. UMC is an on-the-fly analysis framework which allows the user to explore interactively the UML state machine, to visualize abstract behavioural slices of it and to perform local model checking of UCTL formulae. UCTL is an action- and state-based branching-time temporal logic [5]. Its syntax thus allows one to specify the properties that a state should satisfy and to combine these basic predicates with advanced temporal operators dealing with the actions performed.

Layouni et al. in [6] experimented with the use of the model checker Alloy [4] to support policy conflict resolution. In view of its current widespread use in industrial practice, we considered UML a better candidate: it has good tool support, and, besides supporting conflict detection, will also help in understanding and resolving them.

The results reported here were obtained using a prototype translator to automate the translation from the APPEL rules to an equivalent UMC specification, dubbed *Appel2UMC*, and written in OCaml. *Appel2UMC* is structured as a syntax definition module, a *Compiler*, and an *Unparser*. The *Compiler* translates APPEL to UMC, at the abstract syntax level, and the *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, and translation times are not an issue.

In our experiment we considered merging two independently developed managers taking care respectively of performance and power management concerns (part of these rules were introduced in [2]). The performance manager has a rule stating that in case of poor throughput the parallelism degree may be increased. The power manager has a rule stating that in case of too high power consumption the parallelism degree may be decreased. Both managers operate on the application graph executing actions from a set including "LinkWorker" and "UnLinkWorker", including or removing a worker node in/from the current computation, respectively. These link/unlink actions are marked as "atomic conflict" as they clearly negate one another.

[2] 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. Ideally this part would also be executed automatically.

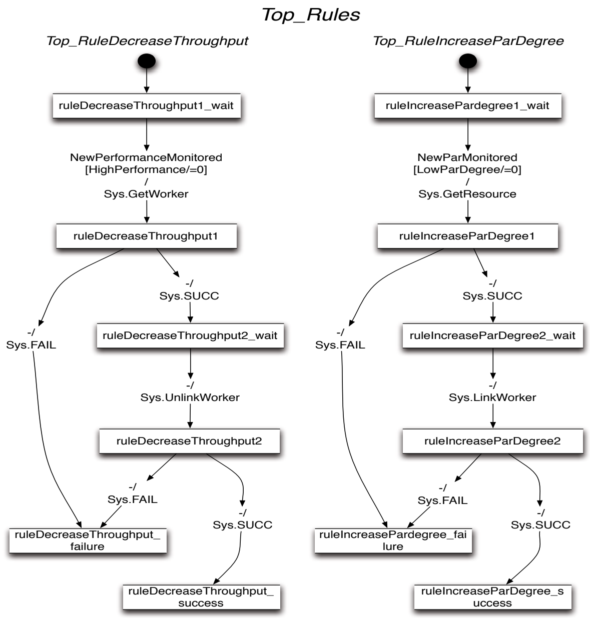
Top_Rules
Top_RuleDecreaseThroughput
Top_RuleIncreaseParDegree
ruleDecreaseThroughput1_wait
NewPerformanceMonitored [HighPerformance/=0] / Sys.GetWorker
ruleDecreaseThroughput1
-/ Sys.SUCC
-/ Sys.FAIL
ruleDecreaseThroughput2_wait
-/ Sys.UnlinkWorker
ruleDecreaseThroughput2
-/ Sys.FAIL
-/ Sys.SUCC
ruleDecreaseThroughput_failure
ruleDecreaseThroughput_success
ruleIncreasePardegree1_wait
NewParMonitored [LowParDegree/=0] / Sys.GetResource
ruleIncreaseParDegree1
-/ Sys.SUCC
-/ Sys.FAIL
ruleIncreaseParDegree2_wait
-/ Sys.LinkWorker
ruleIncreaseParDegree2
-/ Sys.FAIL
-/ Sys.SUCC
ruleIncreasePardegree_failure
ruleIncreaseParDegree_success

Fig. 2. Model checker output: UML parallel state machine

The result of the compilation of the parallel composition of the power and performance manager rules is the UML parallel state machine in Figure 2, whose graphical representation is produced by the UMC framework. In this simple example it is clear, by inspection, that the conflict will arise. To detect it automatically, however, we load the model into the model checker, together with the formalization in UCTL of the relevant question: may a conflict occur in one MAPE cycle? In terms of traces: is there no trace among those generated by the automaton, which includes both link and unlink? Formally:

```
(not EF EX{LinkWorker} EF{UnlinkWorker} true) & (not EF EX{UnlinkWorker} EF{LinkWorker} true)
```

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. The traces of the automaton are shown in Figure 3.

The answer given by the model checker is "false" and the explanation outlines the traces leading to the situation where the formula is demonstrated false. To solve the conflict, the user can then also reason on the UML state machine, which is more expressive than the graph of the traces, including the names of the states and complete labelling of the transitions.

According to the methodology outlined in Sec. 4 we are able to collect the knowledge necessary to produce a modified set of rules that avoid the conflict from the traces exposed by the model checker. From the traces in Fig. 3 we can evince that:

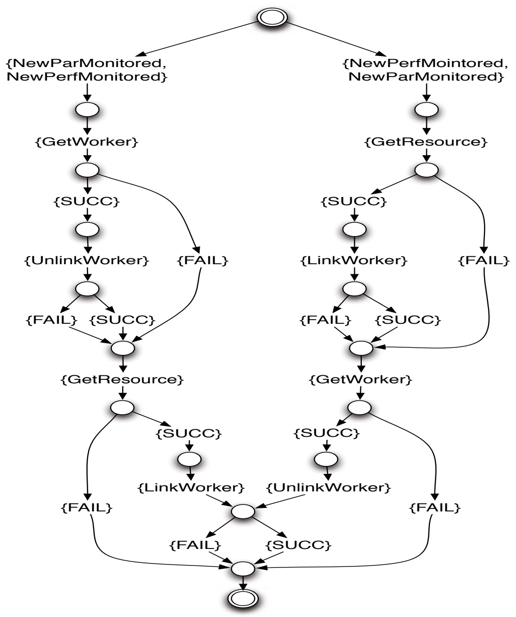

Fig. 3. Model checker output: Traces

- the situation leading to the conflicting actions is determined by the presence of both triggers firing the power manager "reduce power usage" and the performance "increase parallelism degree" rules. This is evidenced by the triggers at the beginning of the two traces.
- both paths leading to the conflict (relative to different interleavings of the same actions) include the actions in the "reduce power usage" and the "increase parallelism degree" rules.

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):

- that includes both triggers[3]; 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 higher priority rule is executed.

This is a very simple case. We modelled just two rules and so we get a very compact model and useful "explanations" in terms of traces. In fact, 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.

[3] possibly a new trigger logically corresponding to the conjunction of the two triggers, as APPEL does not support the conjunction of triggers but only trigger disjunction.

However, we also made more realistic experiments with a set of up to 6 rules with complex action parts.

Fig. 4 shows times needed to execute the model checker with different rules sets and queries (the AG(true) query gives the upper bound in execution times, as it requires the model checker to visit all possible paths in the model). This confirmed to us that the approach is feasible in more realistic situations. We do not show sample output from the model checker in this case, as the graphs are significantly larger and do not fit easily on a page.

Rules#	**∃conflict**	**AG(true)**
2	0.03	0.02
4	0.05	0.12
6	0.06	0.25

Fig. 4. Execution times (in seconds) with different sets of rules and queries

6 Future Work and Conclusions

This paper builds on previous results in the field of multiple non-functional concern management. To the best of our knowledge, the classification of the possible approaches for autonomic management of multiple non-functional concerns presented in Sec. 3 is original. The proposal for using model checking to support merging of independently developed autonomic managers is also new.

In previous work [9] some of the authors presented a compositional, but manual, translation from APPEL to UML and then to UMC. In this paper we automate the translation and introduce a shortcut for those situations where starting from a UML graphical presentation of the rules is not a requirement. Moreover, if we wish to build a transformer from a UML state machine, as generated by a design environment, to a checkable UMC model, we can reuse at least the *Unparser*.

We are currently improving the methodology outlined in this paper. In particular, we are performing more experiments with the model checker to refine the technique (both in the design of the queries and in the interpretation of the analysis) and we are using more realistic rule sets to check that the approach remains feasible.

Acknowledgements. We thank Franco Mazzanti for his support with the model checker. This work is partially supported by Italian PRIN project "SOFT".

References

[1] Aldinucci, M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D., Tonellotto, N.: Behavioural skeletons for component autonomic management on grids. In: Making Grids Work, CoreGRID, Chapter Component Programming Models, pp. 3–16. Springer (August 2008)

[2] Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic managenemt of multiple non-functional concerns in behavioural skeletons. In: Grids, P2P and Services Computing (Proc. of the CoreGRID Symposium 2009), CoreGRID, pp. 89–103. Springer, Delft (2010)

[3] Aldinucci, M., Danelutto, M., Kilpatrick, P., Xhagjika, V.: LIBERO: A Framework for Autonomic Management of Multiple Non-functional Concerns. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 237–245. Springer, Heidelberg (2011)
[4] Alloy Community, `http://alloy.mit.edu/community/`
[5] 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)
[6] Layouni, A., Logrippo, L., Turner, K.: Conflict Detection in Call Control using First-Order Logic Model Checking. In: Proceedings International Conference on Feature Interactions in Software and Communication Systems (ICFI 2007), pp. 66–82. IOS Press (2007)
[7] Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based Conflict Detection for Distributed Policies. Fundamenta Informaticae 89(4), 511–538 (2008)
[8] ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming 76, 119–135 (2011)
[9] ter Beek, M.H., Gnesi, S., Montangero, C., Semini, L.: Detecting policy conflicts by model checking uml state machines. In: ICFI 2009, pp. 59–74 (2009)
[10] Turner, K.J., Reiff-Marganiec, S., Blair, L., Campbell, G.A., Wang, F.: APPEL: An Adaptable and Programmable Policy Environment and Language. Technical Report CSM-161, Univ. of Stirling (2011), `http://www.cs.stir.ac.uk/~kjt/techreps/pdf/TR161.pdf`
[11] UMC v3.7, `http://fmt.isti.cnr.it/umc`