Detecting policy conflicts by model checking UML state machines[1]

Maurice H. TER BEEK [a], Stefania GNESI [a], Carlo MONTANGERO [b] and Laura SEMINI [b,2]

[a] *ISTI–CNR, Pisa*

[b] *Dipartimento di Informatica, Università di Pisa*

Abstract. Policies are convenient means to modify system behaviour at run-time. Nowadays, policies are created in great numbers by different actors, ranging from system administrators to lay-users. However, this situation may lead naturally to inconsistencies, a problem that has been recognized and termed *policy conflict*.

The adoption of a widely-used notation, with good tool support, to express the policies, can not only support the detection, but also help all the involved actors in understanding and resolving the conflicts. In this respect, a natural candidate is UML due to its current wide use in the industrial practice. In this paper we show how to model check policies expressed in UML to verify whether they are free of conflicts: we define a correspondence between APPEL policies and UML state machines and use UMC as a model checker. We validate the approach with examples taken from the literature.

Keywords. Policy conflict, UML, Model checking

Introduction

Policies have been recognized as a convenient means to add flexibility to software systems, since they allow adapting the behaviour at run-time. They were originally introduced in telecommunication systems [1] and have been traditionally applied to access- and usage- control [2,3] and to system management [4,5,6,7]. Recently, policies are being introduced in new domains, like in the control of smart houses [8] and in the automation of business rules in business process management [9]. As a consequence of the broader usage, nowadays policies are created in great numbers by many different actors, ranging from system administrators to lay-users, making worse than ever a problem that has been recognized since the beginnings and termed *policy conflict*, i.e. the fact that policies may be inconsistent and contradict each other.

The issues related to policy conflicts, i.e. conflict definition, detection and resolution are receiving much attention. In particular, a recent trend exploits the current advances in formal static analysis by theorem proving and model checking, applying these tech-

[1]All the authors are partially supported by the EU project SENSORIA IST-2005-16004. The first two authors are also funded by the MIUR–PRIN 2007 project D-ASAP.

[2]Corresponding Author: Laura Semini. Email: semini@di.unipi.it.

niques to anticipate conflict detection—traditionally performed at run-time—at design-time. Indeed, the well-known advantages of early verification apply to policies as well.

The need for wide experimentation with many languages and tools has been recognized by Layouni et al. in [10], where they exploit the model checker Alloy [11] for automated conflict detection. For its current wide use in the industrial practice, we consider UML a natural candidate in this direction. Indeed, adopting a widely used notation, with good tool support, to express the policies, besides supporting conflict detection, will also help all the involved actors in understanding and resolving them. Here, we show how to model check policies expressed in UML for conflict freeness.

Specifically, to build on well-established concepts, we show how to express APPEL policies in UML by defining a correspondence between APPEL and a restricted form of UML state machines (Sect. 1 and 3). Then we discuss how to formulate conflict detection in UMC [12,13]. UMC is a model checker built to analyze UML state machines for properties expressed in the action- and state-based branching-time temporal logic UCTL [14] (Sect. 4). UMC not only allows verifying a system's architecture, but also its behaviour, and like Alloy a counterexample is produced in case of a negative outcome. We validate the approach with a business process management case study (Sect. 2 and 5).

1. APPEL

Policies have been used for some time to adapt the behaviour of systems at run-time. Mostly they have been used in the context of Quality of Service and Access Control. There are a number of policy languages specific to these domains. The APPEL policy language [15,1] has been developed in the context of telecommunication systems, to express end-user policies. A detailed presentation of this language can be found in [16].

APPEL is a general language for expressing policies in a variety of application domains with a clear separation between the core language and its specialization for concrete domains (e.g. telecommunications). Here we concentrate on the core language, whose semantics has been formally defined [17,18] in such a way to maintain such separation, as well as the translation to UML presented in Sect. 3.

A policy consists of a number of policy rules. Policy rules may be grouped using a number of operators (**seq**uence, **par**allel, and **g**uarded and **u**nguarded choice); we will discuss their details in Sect. 3. A policy rule has the following syntax

$$[\textbf{when}\ \mathit{trigger}]\ [\textbf{if}\ \mathit{condition}]\ \textbf{do}\ \mathit{action} \qquad (1)$$

i.e. it consists of an optional trigger, an optional condition and an action. The core language defines the structure but not the details of these parts, which are defined in specific application domains. 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.

The applicability of a rule depends on whether its trigger has occurred and whether its conditions are satisfied. Triggers are caused by external events. Triggers may be combined using **or**, with the obvious meaning that either is enough to apply the rule. Conditions may be combined with **and**, **or** and **not** with the expected meaning. A condition expresses properties of the state and of the trigger parameters. Finally, actions have an effect on the system in which the policies are applied. A few operators (**and**, **andthen**, **or** and **orelse**) have been defined to create composite actions; again, we will discuss their details in Sect. 3. Some examples of policies are given in the next section.

2. Case Study

As a case study we consider a loan negotiation process, provided by the German S&N, an industrial partner of the EU–IST–GC2 project SENSORIA [19]. The example is small and simple, since the purpose is to illustrate the whole process, from the specification, to the conflict detection and resolution. A customer can use a web portal to ask for a loan to a bank. Requests will be forwarded to and handled by the respective local branch, i.e. the one closest to the customer's residence. At the local branch, to process the loan request, and before a contract proposal is sent to the customer, there are two necessary steps: a preliminary evaluation, that we call *vetting*, to ensure that the customer is credible, and a subsequent step, called *approval*, where the contract proposal is approved. The process is extremely simple and rather generic: it only shows two stages (these are essential in the process as specified by the bank to ensure transparency). However, we can imagine a number of refinements that adapt the process to given situations. It is these that we express as policies, and here are some examples:

P1: In a big branch, the request should be vetted and approved by members of the staff working in two distinct offices.
P2: In a small branch, the branch manager has to approve all applications.
P3: In case of loans of small amount, both vetting and approving are done automatically.
P4: If the branch manager of a small branch is out of office, loan applications are approved by the manager representative.

To proceed to the actual APPEL policy definition, we consider `request_submitted` as the trigger for the loan request processing.

```
P1:  when request_submitted if big_branch
     do ( office_A_vetting andthen office_B_approval )
        or ( office_B_vetting andthen office_A_approval )
P2:  when request_submitted if small_branch
     do vet andthen manager_approval
P3:  when request_submitted if small_amount
     do automatic_vetting andthen automatic_approval
P4:  when request_submitted if ( small_branch and unavailable_manager )
     do vet andthen manager_representative_approval
```

3. UML4APPEL

The purpose of this section is to present 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.

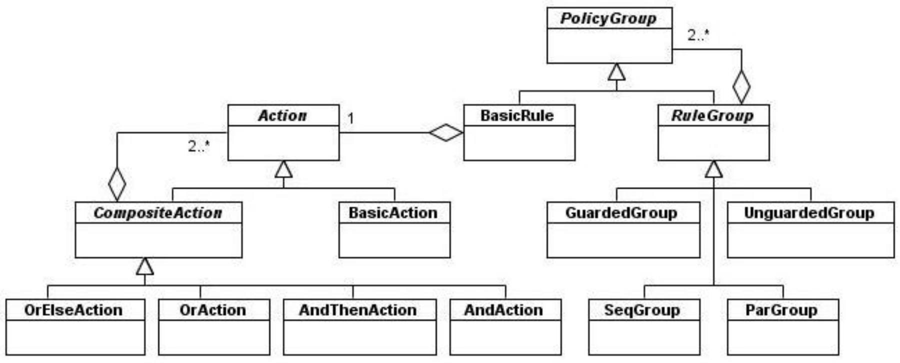

Figure 1. Stereotypes for APPEL.

The target set of UML state machines can be characterized by a *profile*, as is usual when specializing UML to a specific purpose. The APPEL profile defines a hierarchy of *stereotypes* on the UML *states*, as shown in Fig. 1. This hierarchy mimics the syntactic structure of the language, but for the triggers and the conditions. Indeed, triggers and conditions have no graphical counterpart in our mapping, since they map onto the UML triggers and conditions that decorate the machine state transitions, in the natural way. This is fortunate, since triggers and conditions are domain dependent, and we can exploit the flexibility that UML provides w.r.t the language in which to express them, to best fit the domain peculiarities. Basic actions are domain dependent too: the name suffices to characterize them for our purposes. Note that the stereotypes defined as abstract in Fig. 1 (i.e. those in italic) will only appear as placeholders in the mapping rules.

Before presenting the mapping, we need to point out that the action combinators are defined in terms of the outcome of the actions under composition, but in a very broad sense that need not consider the details of the action semantics, but only an abstract notion of *success* and *failure*. Intuitively, these notions entail that an action may complete normally (success) or may abort for some reason (failure). Again, APPEL leaves the specifics of when an action succeeds or fails to the domain, and simply defines the success or failure of a composed action as a combination of the successes and failures of the actions under composition.

*3.1. UML4*APPEL*: Policy Rules*

The upper-left side of Fig. 2 presents the mapping of a basic rule: the machine is parametric w.r.t. *RuleName*, *ts*, *cs* and *a*, where ts is a trigger, cs is a condition, and a is an action, according to the syntax in (1). In particular, the rule can be either basic or composed, according to the patterns given next.

It may be useful to recap the behaviour of a «basicRule» machine: it starts in state w, waiting for an event matching its triggers ts. On the occurrence of such an event, if (and only if) condition cs holds in the current state of the system subject to the policies, the machine transits into «action» state a. From a UML point of view, this is a state of type activity, i.e. one in which something is carried out. In this profile, states are either basic or composed «action»s. The action can terminate in one of the exit states: the one with stereotype «s» if the action terminates successfully, the other one («f») if it fails. In either case, the rule as a whole terminates.

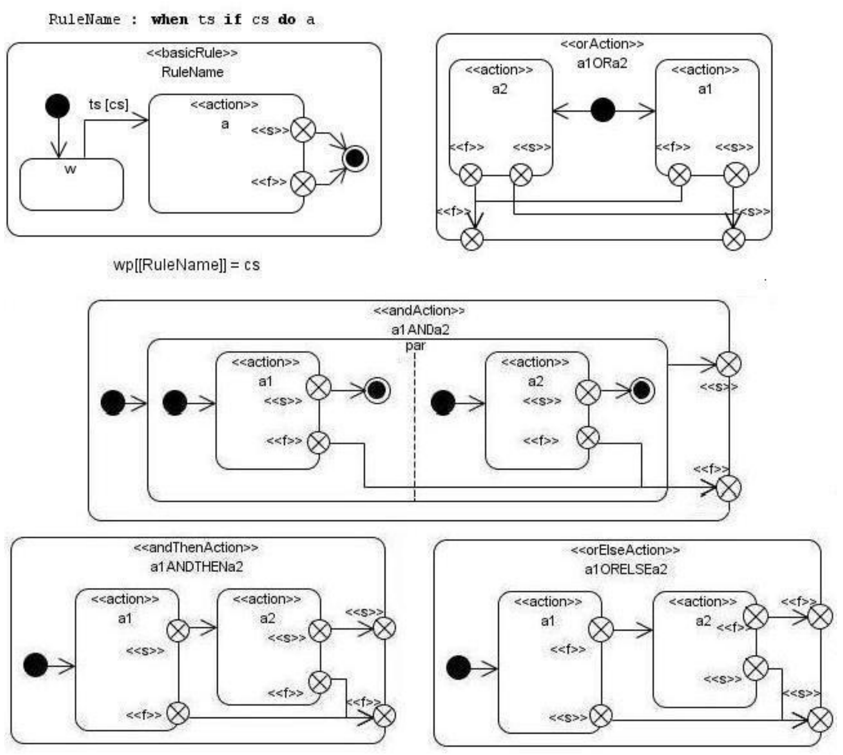

Figure 2. Basic rule (top left), and composing actions: **or** (top right), **and** (middle), **andthen** (bottom left), and **orelse** (bottom right).

The annotation on the upper-left machine of Fig. 2 defines the *Weakest Precondition* of the rule as the *conditions* in the rule: its meaning will become clear when we compose rules in Sect. 3.2. Next, we discuss action composition: we present a translation rule per each action composition operator.

a1 **and** *a2* The middle part of Fig. 2 shows the composition pattern for *a1* **and** *a2*. This operator has the most complex semantics: both actions are started in parallel (hence the two regions in the composite state *a1ANDa2*), and the composed action succeeds only if both *a1* and *a2* succeed. Otherwise, as soon as one action fails, the other is forced to fail, and the composed action fails.

a1 **or** *a2* In this case (see Fig. 2, top right) only one action is chosen nondeterministically[3] and the outcome of the composed action follows that of the chosen action.

a1 **andthen** *a2* Action *a1* is executed first: if it succeeds, action *a2* is also executed. Then, if *a2* also succeeds, the composed action succeeds too. In any other case the composition fails, as shown in Fig. 2 (bottom left).

a1 **orelse** *a2* This is the dual of the previous one: *a2* is attempted only if *a1* fails, and the composition fails only if both *a1* and *a2* fail (Fig. 2, bottom right).

[3]In UML, what happens when two transitions can fire, as happens in the initial state here, is a semantic variation point: we assume a nondeterministic choice, as it is in UMC.

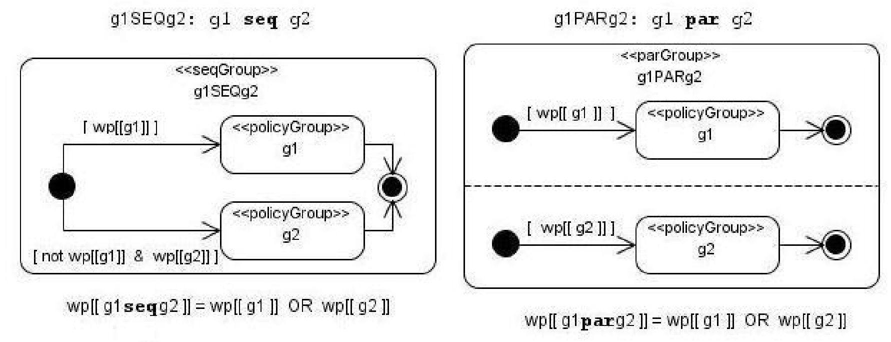

Figure 3. Rule groups: **seq** (left) and **par** (right).

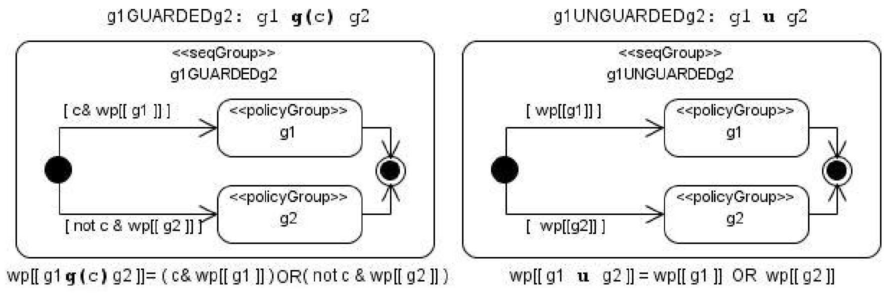

Figure 4. Rule groups: guarded and unguarded choice.

*3.2. UML4*APPEL *: Policy Rule Groups*

The first step in defining the policy groups in UML is to understand how the group operators work. Essentially, they order the triggering of the rules according to their applicability, i.e. according to the value of the conditions that allow them to fire. In $g1$ **seq** $g2$, e.g., the applicability of $g2$ is considered after that of $g1$ (the name of the operator, **seq**, stands for sequence of consideration, i.e. priority): in the presence of events triggering both $g1$ and $g2$, $g2$ is considered only if $g1$ is not applicable. Similarly, if $g1$ is applicable, but there are no events to trigger it, no rule will fire, even if $g2$ is applicable and there are events able to trigger it. The applicability conditions can be computed recursively as wp (*weakest preconditions*) when grouping rules, starting from the basic rule r, where $wp[[r]] = cs$, as shown in Fig. 2 (top left). These definitions are given in the figures discussed in the sequel.

name: g1 **seq** *g2* The left part of Fig. 3 describes how either *g1* or *g2*—but not both—can become active, according to the above discussion. Then, the active one may fire in the presence of suitable events. To consider the group applicable as a whole, at least one of the guards on the initial transitions has to be verified, hence their disjunction in the definition of the wp of the group, below the machine.

name: g1 **par** *g2* As shown in the right part of Fig. 3, the two groups are simply put in parallel. The composite group is applicable if one or both groups are. Hence, the wp of the composite is the disjunction of the component wp's.

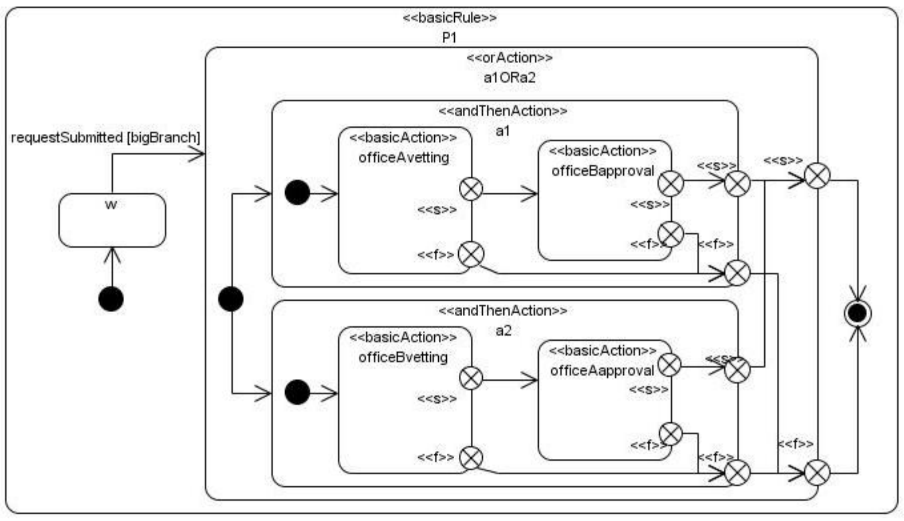

Figure 5. State machine for policy P1.

name: g1 **g**(c) *g2* As shown in the left side of Fig. 4, the first group is executed if the condition in the guard is satisfied and the group is applicable. Otherwise, the second group is executed, if it is applicable.

name: g1 **u** *g2* In this case (right side of Fig. 4), if only one group is applicable, it is executed. If both are applicable, one is chosen nondeterministically.

As an example of application of these rules, reconsider the case study: the UML representation of policies P1–P4 is given in Figs. 5 and 6.

3.3. Conflict Definition and Detection

We exploit UML to define conflicts between actions, too. We illustrate this point using the loan process example of Sect. 2. We model the domain of actions using hierarchies, with the generalization relation. In the upper part of Fig. 7, we say that *vetting* can be automatic or manual and, in the latter case, it can be carried out by office A or B. Similarly, the lower part of the figure defines the model for *approval*. We also assume that the root and the intermediate nodes in the hierarchy are abstract, and only the leaves are concrete actions: a policy requiring an abstract action `a` actually requires any action which is a leaf of the subtree rooted in `a`, e.g. `manual_vetting` means `office_A_vetting` or `office_B_vetting`.

This way of modelling the action domain naturally leads to the following construction: actions belonging to different paths in a tree are in conflict; different actions in the same path are possibly in conflict (a conflict is possible, due to disjunction). This way, a conflict set of action pairs is automatically derived. The specifier can then refine this set by removing non-conflicting leaves, or by adding conflicting actions in separate trees.

To detect conflicts, we anticipate that in UMC, the behaviour of a state machine is formalized as a labelled transition system (see Sect. 4.2). It is natural to require that no execution path includes a pair of conflicting actions, in any order. More sophisticated properties can be expressed too. We could, e.g., forbid executions where automatic approval follows manual vetting.

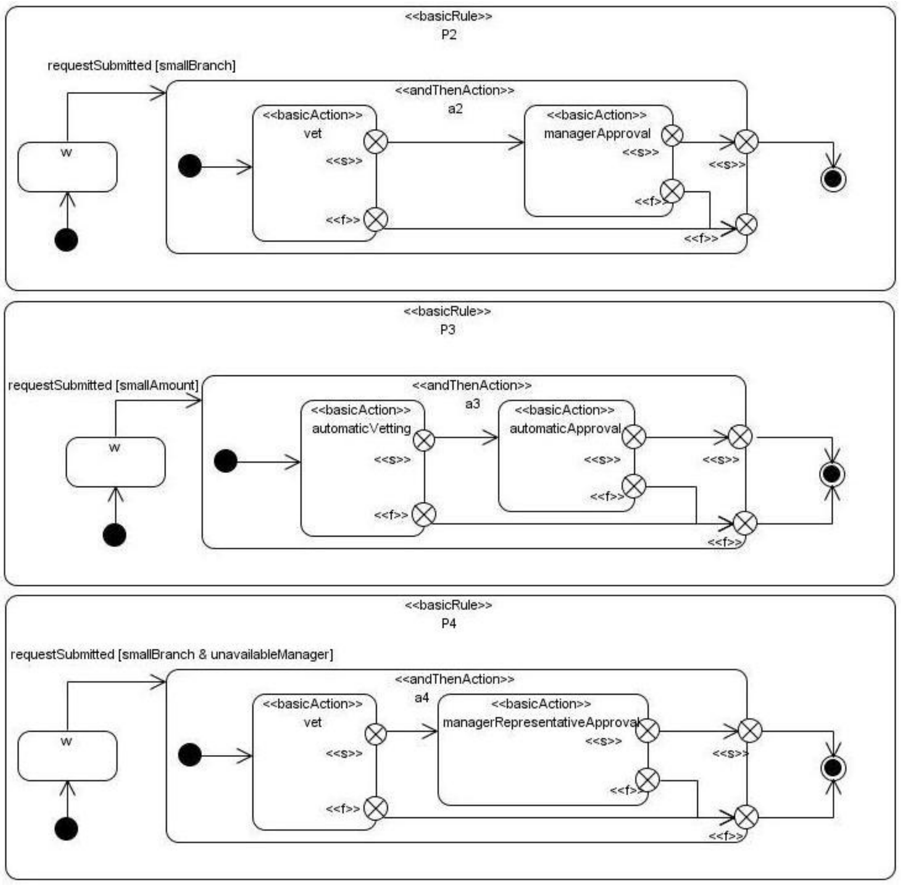

Figure 6. State machine for policy P2 (top), P3 (middle) and P4 (bottom).

4. Model Checking UML State Machines

In order to use the full potential of specification languages like UML, which allow action as well as state changes to be modelled, several verification techniques for the validation of behavioural properties over one's (UML) model have been introduced. In this paper, we will use the on-the-fly model checker UMC [12,13] and the associated action- and state-based branching-time temporal logic UCTL [14].

Model checking is an automatic technique for verifying correctness properties of system designs [20]. Such verifications are exhaustive, i.e. all possible input combinations and states are taken into account, and a counterexample is usually generated in case a certain property does not hold. Correctness properties reflect typical (un)desired behaviour of the system under scrutiny. The system is usually given as a description in a special-purpose language. Such a description corresponds to a Labelled Transition System (LTS), i.e. a directed graph consisting of nodes and labelled edges. The nodes represent system states, the edges (labelled with actions) represent possible transitions which may alter the state (due to an action being performed). Usually, a set of atomic propositions is associated with each node to represent the basic properties that hold at a point of execution. Hence, formally, the problem of model checking can be stated as follows:

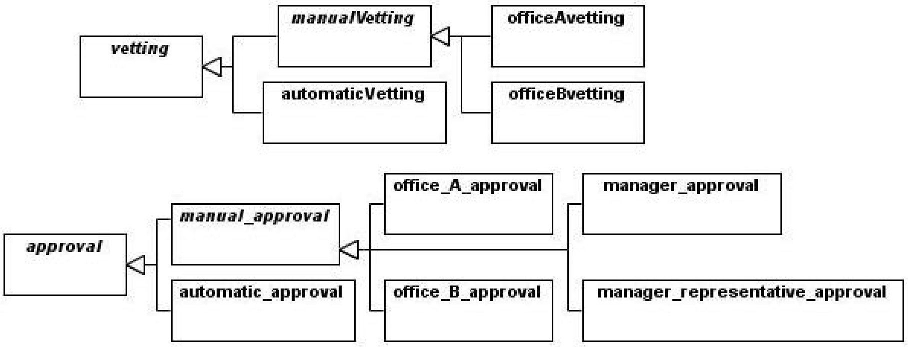

Figure 7. Action hierarchies.

given a desired property, expressed as a temporal logic formula f, and a structure M with initial state s, decide whether $M, s \models f$, where $\models$ is a satisfaction relation. If M is finite, model checking thus reduces to a graph search.

4.1. The Temporal Logic UCTL

Temporal logics allow reasoning about a time line (a path in a graph). Linear-time logics are restricted to this type of reasoning, whereas branching-time logics can reason about multiple time lines. UCTL includes both the branching-time action-based logic ACTL [21] and the branching-time state-based logic CTL [20]. 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. We consider here the following fragment of UCTL:

$$\phi ::= \mathit{true} \mid p \mid \phi \wedge \phi' \mid \neg\phi \mid E\pi \mid A\pi$$
$$\pi ::= X_\chi\, \phi \mid \phi\ {}_\chi U_{\chi'}\ \phi'$$

Predicates are ranged over by p, *state formulae* are ranged over by ϕ, *path formulae* are ranged over by π and *actions* are ranged over by χ. E and A are the *path quantifiers* “exists” and “for all”, while X and U are the indexed *temporal operators* “next” and “until”. The “next” operator X says that in the next state of the path, reached by an action formula satisfying χ, formula ϕ holds. The intuitive meaning of the doubly-indexed “until” operator U on a path is that ϕ' holds at some future state of the path reached by a last action formula satisfying χ, while ϕ has to hold from the current state until that state is reached and all actions executed in the meantime along the path either satisfy χ or τ.

Starting from these basic UCTL operators, it is straightforward to derive the standard logical operators $\vee$ and $\Rightarrow$, well-known temporal operators such as EF (“possibly”), AF (“eventually”) and AG (“always”), and the diamond and box modalities of the Hennessy-Milner logic [22].

The semantic domain of UCTL is a doubly-labelled transition system (L^{2}TS for short) [23]. An L^{2}TS is an LTS whose nodes are labelled by atomic propositions and whose transitions are labelled by sets of actions.

4.2. The Model Checker UMC

UMC is an on-the-fly model checker allowing the efficient verification of UCTL formulae (i.e. specifying action- and/or state-based properties) over a set of communicating UML state machines [24]. The possible system evolutions are formally represented as an L^2TS, whose nodes represent the various system configurations and whose transitions represent the possible evolutions of a system configuration. More concretely, the nodes of this L^2TS are labelled with the observed structural properties of the system configurations (like the active substates of objects, the values of object attributes, etc.), while its transitions are labelled with the observed properties of the system evolutions (like which is the evolving object, which are the executed actions, etc.).

The basic idea underlying UMC is that, given a state of an L^2TS, the validity of a UCTL formula on that state can be evaluated by analyzing the transitions allowed in that state and the validity of a certain subformula in some of the next reachable states, all this in a recursive way. It is due to the on-the-fly verification technique—which verifies a system without actually building a model of it, in the sense that the construction of the state space (L^2TS) is done partially while the verification proceeds without storing states that were already searched—that only some of the next reachable states are searched. UMC uses an on-the-fly model-checking algorithm which has a linear complexity in the size of the formula and the dimension of the L^2TS.

The current UMC prototype (v3.6) can be experimented via a web interface [13]. Due to its reasonably low memory requirements, UMC v3.6 can easily handle state spaces of upto 1,000,000 states, which could not possibly be analyzed by hand.

5. Case Study in UMC

UML state machines are a widely used specification language to describe, in an object-oriented way, the behaviour of (concurrent, distributed) systems. UML state machine diagrams are the standard formalism to design state machines, and many tools exist to support them. In UMC a state machine diagram (modelling a state machine) is associated to the notion of *class*, while a system's configuration is defined by a set of *objects* (active class instances). Classes define the dynamic behaviour of their objects (through state machine diagrams), together with their interfaces towards other components (i.e. the set of signals and operations they are able to receive) and the structure of the local status of the object instances (i.e. the local object attributes).

The underlying L^2TS of a UMC specification is defined by the informal operational semantics of a UML system constituted by a set of concurrent communicating state machines, in which the parallelism among objects is modelled through interleaving, the communications are modelled as immediate and failure-free, and the event queues are plain FIFO queues. The edges of a L^2TS are labelled by the events of sending signals between state machines, while its nodes are used to denote that a certain state of an object is currently active or that a certain attribute of an object has a certain value.

5.1. Conflict Detection

We return to our case study to show how UMC can be used to detect policy conflicts. As an example we check whether policies P2 and P4 put in parallel are conflict-free.

```
Class System is                                          -- class definition
Signals: requestSubmitted                                -- public interface
State S = S1, S2                                         -- private part
Transitions: S1 -> S2 {- /P2parP4.requestSubmitted}
end System;

Class Policies is                                        -- class definition
Signals: requestSubmitted                                -- public interface
Vars: smallBranch:bool, managerUnavailable:bool,         -- private part
      managerApproval:bool, managerRepresentativeApproval:bool;
State P = Par
State Par = P2 / P4                                      -- P2 in parallel with P4
State P2 = w, a2, f2
State a2 = vet, managerApproval
State P4 = w, a4, f4
State a4 = vet, managerRepresentativeApproval
Transitions:
 P2.w -> a2 {requestSubmitted [smallBranch=true]}
 a2.vet -> managerApproval {-/ managerApproval:=true}
 a2.vet -> f2
 a2.managerApproval -> f2
 P4.w -> a4 {requestSubmitted [smallBranch=true & managerUnavailable=true]}
 a4.vet -> managerRepresentativeApproval {-/ managerRepresentativeApproval:=true}
 a4.vet -> f4
 a4.managerRepresentativeApproval -> f4
end Policies;

Objects:                                                 -- static object instantiation
Sys: System
P2parP4: Policies (smallBranch -> true, managerUnavailable -> true,
         managerApproval -> false, managerRepresentativeApproval -> false)
```

Figure 8. UMC specification of system and P2 and P4 in parallel.

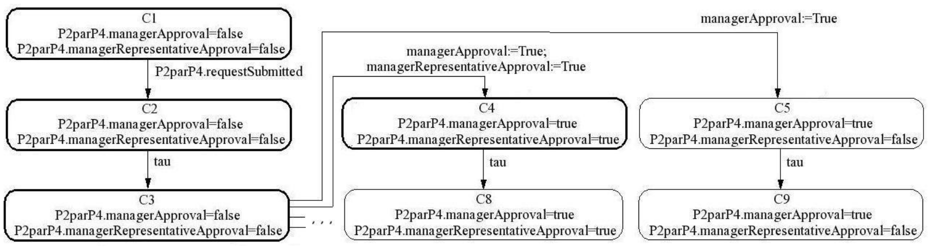
C1
P2parP4.managerApproval=false
P2parP4.managerRepresentativeApproval=false
P2parP4.requestSubmitted
C2
P2parP4.managerApproval=false
P2parP4.managerRepresentativeApproval=false
tau
C3
P2parP4.managerApproval=false
P2parP4.managerRepresentativeApproval=false
managerApproval:=True
managerApproval:=True;
managerRepresentativeApproval:=True
C4
P2parP4.managerApproval=true
P2parP4.managerRepresentativeApproval=true
C5
P2parP4.managerApproval=true
P2parP4.managerRepresentativeApproval=false
tau
tau
C8
P2parP4.managerApproval=true
P2parP4.managerRepresentativeApproval=true
C9
P2parP4.managerApproval=true
P2parP4.managerRepresentativeApproval=false

Figure 9. State space of system with policies P2 and P4 in parallel.

Since both P2 and P4 have `request_submitted` as trigger, these policies can be applied only after the underlying system has sent this signal. We therefore abstract from details of the system, but specify it in UMC as a class `System` that sends the signal `requestSubmitted` to the parallel composition of P2 and P4, defined itself as class `Policies`. We then initiate an object `Sys` of class `System` and an object `P2parP4` of class `Policies`. The UMC code of this specification is given in Fig. 8, while its full state space is depicted in Figure 9. Finally, Fig. 10 (left) shows the UMC representation of state machines P2 and P4 in parallel that UMC has produced from the code of Fig. 8.

We now use UMC to detect possible conflicts. According to the definition of pairs of conflicting actions given in Sect. 3.3, the actions `manager_approval` and `manager_representative_approval` are in conflict since they occur on different branches in the approval action hierarchy given in Fig. 7. We use UMC to check whether the full state space of this specification given in Fig. 8 contains a path along which both actions are executed (in any order). We formalize this in UCTL as follows:

$$AG\ \neg((\text{managerApproval}=true)\ \&\ (\text{managerRepresentativeApproval}=true)) \qquad (2)$$

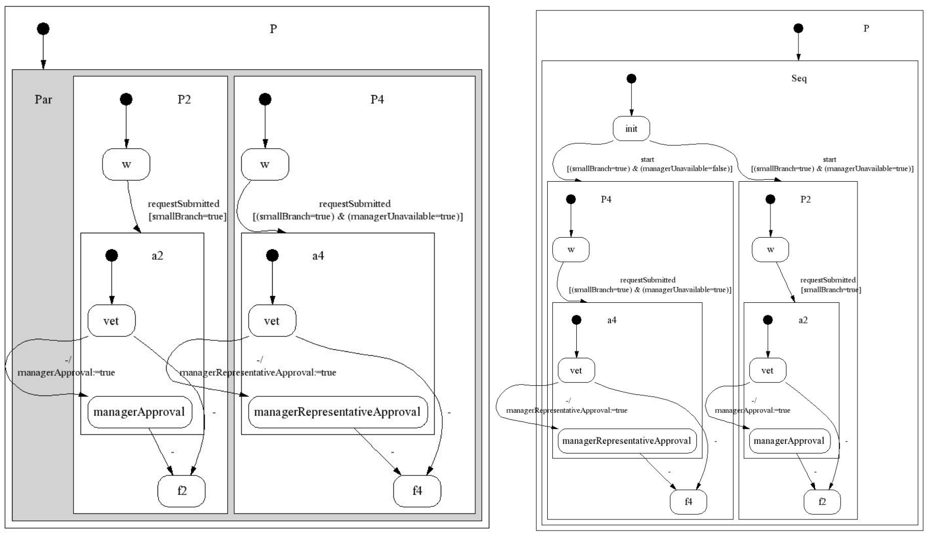
P
Par
P2
P4
w
w
requestSubmitted
[smallBranch=true]
requestSubmitted
[(smallBranch=true) & (managerUnavailable=true)]
a2
a4
vet
vet
-/
managerApproval:=true
-/
managerRepresentativeApproval:=true
managerApproval
managerRepresentativeApproval
-
-
-
-
f2
f4
P
Seq
init
start
[(smallBranch=true) & (managerUnavailable=false)]
start
[(smallBranch=true) & (managerUnavailable=true)]
P4
P2
w
w
requestSubmitted
[(smallBranch=true) & (managerUnavailable=true)]
requestSubmitted
[smallBranch=true]
a4
a2
vet
vet
-/
managerRepresentativeApproval:=true
-/
managerApproval:=true
managerRepresentativeApproval
managerApproval
-
-
-
-
f4
f2

Figure 10. Policies P2 and P4 in parallel (left) and in sequence (right).

```
The formula is FALSE (7 system configurations observed)
------------------------------------------------------------------------------------------------
The formula:  AG not ((managerApproval=true) and (managerRepresentativeApproval=true))
is FOUND_FALSE in State C1 because
  C1 --"Obj:requestSubmitted"--> C2
  C2 --"Obj:"--> C3
  C3 --"Obj:managerApproval:=True;managerRepresentativeApproval:=True"--> C4
  The formula:  not ((managerApproval=true) and (managerRepresentativeApproval=true))
  is FOUND_FALSE in State C4
------------------------------------------------------------------------------------------------
The formula:  not ((managerApproval=true) and (managerRepresentativeApproval=true))
is FOUND_FALSE in State C4 because
  The formula:  (managerApproval=true) and (managerRepresentativeApproval=true)
  is FOUND_TRUE in State C4
------------------------------------------------------------------------------------------------
The formula:  (managerApproval=true) and (managerRepresentativeApproval=true)
is FOUND_TRUE in State C4 because
  The formula: (managerApproval=true)  is FOUND_TRUE in State C4 and because
  The formula: (managerRepresentativeApproval=true)  is FOUND_TRUE in State C4
```

Figure 11. Output of UMC after verifying Formula (2) on the state space in Fig. 9.

Formally, the construct $AG\ f$ holds in a state s if and only if f holds in s and in all the states that are reachable from s in one or more steps (along any path). Hence, Formula (2) intuitively says that “no path may exist, along which actions `managerApproval` and `managerRepresentativeApproval` are both executed”.

Asked to verify Formula (2), it takes UMC a negligible amount of time to produce the output given in Fig. 11. To do so, UMC’s on-the-fly algorithm only constructs states `C1`–`C4`, i.e. those highlighted in Fig. 9. Since Formula (2) is false, the system composed with policies P2 and P4 in parallel produces a conflict.

In this simple example, the full state space given in Fig. 9 is so small that we could easily have examined it by hand to conclude that Formula (2) is false. However, one can easily imagine that this is no longer the case for much larger systems and much more complex policies or, for that matter, for much more complex formulae.

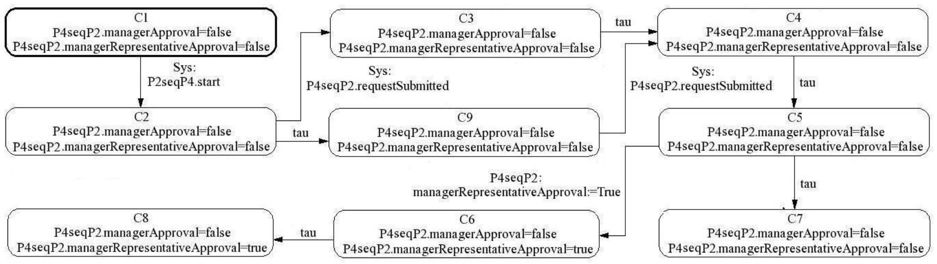
C1
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
Sys:
P2seqP4.start
C2
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
C3
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
Sys:
P4seqP2.requestSubmitted
tau
C9
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
tau
C4
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
Sys:
P4seqP2.requestSubmitted
tau
C5
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false
P4seqP2:
managerRepresentativeApproval:=True
tau
C8
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=true
tau
C6
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=true
C7
P4seqP2.managerApproval=false
P4seqP2.managerRepresentativeApproval=false

Figure 12. Evolutions of policies P4 and P2 in sequence.

```
The formula is TRUE (8 system configurations observed)
-----------------------------------------------------------------------------------------------
The formula:  AG not ((managerApproval=true) and (managerRepresentativeApproval=true))
is FOUND_TRUE in State C1 because
  for all paths starting from here the formula:
    not ((managerApproval=true) and (managerRepresentativeApproval=true))  will continuously hold
```

Figure 13. Output of UMC after verifying (2) on the state space in Fig. 12.

5.2. Conflict Resolution

In an attempt to resolve the above conflict, we change the specification by putting the policies P4 and P2 in sequence rather than in parallel. We do not provide the UMC code of this specification that has produced the UMC representation of state machines P4 and P2 in sequence presented in Fig. 10 (right). Its full state space is depicted in Fig. 12.

Asked to verify Formula (2) for this new specification, it takes UMC a negligible amount of time to produce the output given in Fig. 13. This time UMC is forced to construct the full state space (Fig. 12) to reach this conclusion. Since Formula (2) is now true, we conclude that the system composed with policies P4 and P2 in sequence does not produce a conflict.

We conclude that putting policies P4 and P2 in sequence has successfully resolved the conflict detected for policies P2 and P4 in parallel.

6. Related Work

Conflict detection and resolution have been addressed by many authors. The approaches focus on different intervention times. *Design time*: Amer et al. [25] aim at defining hierarchical policies, so that the subordinate policies cannot conflict, by construction. Similarly, Halpern and Weissman [26] introduce a fragment of first-order logic, more expressive than Datalog, such that conflicts cannot arise. To support the naive users, a friendly interface allows querying the policy set for permissible/prohibited actions. *Run-time*: the use of meta-policies (i.e. policies that act on them) is proposed in [27]; the meta-policies are triggered by the normal ones during execution. In [2], policies are expressed as safety conditions in Interval Temporal Logic, which are checked at run-time by the simulation tool Tempura. *Hybrid approaches*: meta-policies are proposed also in [7]; meta-policy checks are applied when policies are specified, not only when they are executed. Dunlop et al. [4] discuss the need for both static and dynamic conflict detection and resolution, and introduce computationally feasible algorithms to this purpose. They exploit a deontic logic of permission, prohibition and obligation, coupled with temporal classifiers that indicate the span of the mode.

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 more distributed, this will become less feasible as conflicts often can only be dealt with at run-time. Nevertheless, there is a wide spectrum between the two extremes, e.g. when a base set of policies is deployed in a product, and it can be updated by the user while the system is operating. In our approach, new policies can be defined at operation time, and checked for consistency before their actual insertion in the policy set. UML and UMC are the language and tool we propose to this purpose. Then, in this setting, resolution means defining a precedence relation between new and existing rules, using the **seq** construct.

A relevant related topic is feature interaction. Features stem from the telecommunications industry, but similar concepts exist in other areas such as component-based systems. In general a feature is new functionality to enhance a base system. Features are often developed in isolation and each feature's operation is tested w.r.t. the base system, and also with common known features. When two or more features are added to a base system, unexpected behaviour might occur. This is caused by features influencing each other, and is referred to as feature interaction. Feature interaction shows many similarities to policy conflict, the main difference being the detail to which it has been studied. A general discussion of the problem appears in [28].

In the literature, conflict detection and resolution have been addressed first, somewhat assuming that conflict definition was not an issue. This was likely because conflict definition pertains to the application domain, and policies have been traditionally applied in the well-structured domains listed in the Introduction. Recently, however, the issue of conflict definition has received more attention, as policies are being introduced in new domains, like in that of our case study: the automation of business rules in business process management [9].

Shehata et al. investigate the policy conflict problem in the smart homes domain. Policies are a flexible way for users that have to control home devices. However, also in this domain user policies often tend to interact in unwanted ways leading to unexpected behaviour. In [8], Shehata et al. propose the use of IRIS, a semi-formal method for defining and detecting conflicts, based on the systematic production of tables and graphs and on the analysis of these graphs and tables according to given guidelines. A run-time module for detecting and resolving policy conflicts is defined, based on simulation techniques. These studies include also an exhaustive set of examples of interactions between policies in the smart homes domain. Since the smart home is essentially controlled by action and goal policies, it could be interesting to experiment UMC in this setting.

In previous work [9], we introduced STPOWLA, a novel combination of policies (expressed in APPEL) and workflows that allows capturing the essential requirements of a business process (as a workflow) and expressing variability in a descriptive way (as policies). Besides, STPOWLA is SOA based: each task in a workflow is executed by a service, and policies express both the functional and SL requirements of the service. The work presented here builds also on [17,18], where policies and conflicts are defined by liveness formulae in a distributed temporal logic, and detected by (semi-)automatic theorem proving: a conflict is found if it is possible to derive, from the logical presentation of the policies, a formula stating that a conflict will arise.

7. Conclusions and Future Work

In this paper we have shown how to model check policies expressed in UML to verify whether they are free of conflicts. To this aim, we have defined a correspondence between APPEL policies and UML state machines and used UMC as a model checker. We have validated the approach with examples taken from the literature. We thus respond to the need for experimentation with many languages and tools that has been recognized by Layouni et al. in [10], where Alloy [11] is exploited for automated conflict detection. We have chosen UML since it is the de facto industrial standard for system modelling, and because the complexity of using UML and UMC is comparable with other techniques. Future work include the experimentation on larger case studies, and on a larger subset of APPEL, for validating the scalability of our approach, as well as the development of a tool to map APPEL into XMI, for subsequent model checking.

Alloy allows modelling and verifying the architecture of a system for consistency w.r.t. system properties and constraints, expressed as predicates in a relational extension of first-order logic. To perform verification, the model and predicates are translated into a (usually large) boolean formula, whose satisfiability is decided using efficient SAT solvers. This problem is of exponential complexity in the size of the formula. Alloy uses a bounded model-checking algorithm, i.e. it unrolls the state space for a fixed number of steps k and checks whether a property violation can occur in k or fewer steps. Since inconsistencies could in theory manifest themselves for larger searches, this process would need to be repeated with larger and larger values of k until all possible violations have been ruled out. UMC, on the other hand, uses an on-the-fly model-checking algorithm. This has the advantage that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result (i.e. either the formula is valid or a counterexample is produced). This algorithm is moreover of linear complexity in the size of the formula and the dimension of the model.

In [29], Shehata et al. have introduced a taxonomy for policies, distinguishing among *action* policies, i.e. policies that "dictate the action that should be taken whenever the system is in a given state"; *goal* policies, which are at a higher level: rather than specifying the possible action in a given state explicitly, they define a set of desirable states, leaving the system to decide the actions to be taken; *utility function* policies that add still more flexibility: it is possible to assign a utility value to each desirable state.

According to this taxonomy, what we consider in this paper are action policies. We plan to consider, in future work, applying our approach to deal with goal policies. We would assume the system to be modelled as a state machine and express the policies by naming the states to be reached in given circumstances, or by characterizing these desirable states in terms of attribute values with constraints. With the utility function policies, the objective is to maximize utility, and conflict detection is no longer an issue.

References

[1] K.J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland. Policy Support for Call Control. *Computer Standards and Interfaces*, 28(6):635–649, 2006.

[2] F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *Proceedings Formal Methods in Security Engineering (FMSE '03)*, pages 32–42. ACM Press, 2003.

[3] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.

[4] N. Dunlop, J. Indulska, and K. Raymond. Methods for Conflict Resolution in Policy-Based Management Systems. In *Proceedings Enterprise Distributed Object Computing (EDOC'03)*, pages 98–111. IEEE Computer Society, 2003.

[5] J.D. Moffett and M.S. Sloman. Policy Conflict Analysis in Distributed Systems Management. *J. Organizational Computing*, 4(1):1–22, 1994.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In M. Sloman, J. Lobo, and E. Lupu, editors, *Proceedings of Policies for Distributed Systems and Networks (POLICY'01), Bristol, UK*, volume 1995 of *LNCS*, pages 18–38. Springer, 2001.

[7] E. Lupu and M. Sloman. Conflicts in Policy Based Distributed Systems Management. *IEEE Trans. Software Eng.*, 25(6), 1999.

[8] M. Shehata, A. Eberlein, and A.O. Fapojuwo. Using semi-formal methods for detecting interactions among smart homes policies. *Sci. Comput. Program.*, 67(2-3):125–161, 2007.

[9] S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, Policies and Workflows. In *Revised Selected Papers of Workshops International Conference on Service-Oriented Computing (ICSOC'07)*, volume 4907 of *LNCS*, pages 351–362. Springer, 2007.

[10] A.F. Layouni, L. Logrippo, and K.J. Turner. Conflict Detection in Call Control using First-Order Logic Model Checking. In *Proceedings International Conference on Feature Interactions in Software and Communication Systems (ICFI'07)*, pages 66–82. IOS Press, 2007.

[11] Alloy Community. Online: `http://alloy.mit.edu/community/`.

[12] F. Mazzanti. UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, 2006.

[13] UMC v3.6. Online: `http://fmt.isti.cnr.it/umc`.

[14] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In *Revised Selected Papers of Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 133–148. Springer, 2007.

[15] S. Reiff-Marganiec, K.J. Turner, and L. Blair. APPEL: The Accent Project Policy Environment/ Language. Technical Report TR-161, University of Stirling, 2005.

[16] S. Reiff-Marganiec and K.J. Turner. Use of Logic to Describe Enhanced Communication Services. In *Proceedings Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *LNCS*. Springer, 2002.

[17] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based detection of conflicts in APPEL policies. In *Proceedings Fundamentals of Software Engineering (FSEN'07)*, volume 4767 of *LNCS*, pages 257–271. Springer, 2007.

[18] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based Conflict Detection for Distributed Policies. *Fundamenta Informaticae*, 89(4):511–538, 2008.

[19] SENSORIA EU project. Home page: `http://www.sensoria-ist.eu`.

[20] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[21] R. De Nicola and F. Vaandrager. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[22] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32(1):137–161, 1985.

[23] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[24] Unified Modeling Language. Online: `http://www.uml.org/`.

[25] M. Amer, A. Karmouch, T. Gray, and S. Mankovski. Feature Interaction Resolution using Fuzzy Policies. In *Proceedings Feature Interactions in Telecommunications and Software Systems (FIW'00)*, pages 94–112. IOS Press, 2000.

[26] J.Y. Halpern and V. Weissman. Using First-Order Logic to Reason about Policies. In *Proceedings Computer Security Foundations Workshop (CSFW'03)*, pages 187–201. IEEE Computer Society, 2003.

[27] K.J. Turner and L. Blair. Policies and conflicts in call control. *Comput. Networks*, 51(2):496–514, 2007.

[28] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. Feature Interaction: A critical Review and Considered Forecast. *Comput. Networks*, 41:115–141, 2001.

[29] J.O. Kephart and W.E. Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In *Proceedings Policies for Distributed Systems and Networks (POLICY'04)*, pages 3–12. IEEE Computer Society, 2004.