Logic–based detection of conflicts in Appel policies

Carlo Montangero[1], Stephan Reiff-Marganiec[2], and Laura Semini[1]

[1] Dipartimento di Informatica, Università di Pisa
email: {monta,semini}@di.unipi.it
[2] Department of Computer Science, University of Leicester
email: {srm13}@le.ac.uk

Abstract. Appel is a general language for expressing policies in a variety of application domains with a clear separation between the core language and its specialisation for concrete domains. Policies can conflict, thus leading to undesired behaviour. We present a novel formal semantics for the Appel language based on ΔDSTL(x) (so far Appel only had an informal semantics). ΔDSTL(x) is an extension of temporal logic to deal with global applications: it includes modalities to localize properties to system components, an operator to deal with events, and temporal modalities à la Unity. A further contribution of the paper is the development of techniques based on the semantics to reason about conflicts.

1 Motivation

In general the idea of policies is to adapt the behaviour of an existing system. Policies are high-level statements to support personal, organisational or system goals. Policies have been defined as *information which can be used to modify the behaviour of a system* [11]. Policies have been studied in applications such as distributed systems management, network management, Quality of Service, and access control [12, 21].

More recently policies have been studied as means for end-users to express how they want for a system to behave. The work has mostly concentrated on telecommunications systems [22], but there have also been initial attempts at transferring this to service oriented systems [9]. Notably these approaches have been more operational in nature, that is they use a general purpose policy language with an informal semantics.

In general policies are not singular entities, they are generally arranged in groups to collectively express overall goals. However, when several policies are composed (or applied simultaneously) they might contradict each other: a phenomenon referred to as policy conflict. Policy conflict has been recognised as a problem [18] and there have been some attempts to address this, mostly in the domain of access or resource control. In the case of end-user policies the problem is significantly increased by a number of factors. To name a few: the application domains are much more open and hence increase difficulty in modelling them, there will be many more end-user policies than there are system

policies (sheer number of policies) and end-users are not necessarily aware of the wider consequences of a policy that they formulate. However, policy conflict hinders maximum gain when using policies and hence it is important to tackle the problem. Both, detection and resolution are important aspects – resolution at design-time means redesign of the policies.

We propose a logic based reasoning approach to detect policy conflict in the APPEL policy language [19, 22]. As APPEL has so far only been presented with an informal semantics, we must first formalise this in a suitable logic. This paper presents the novel formal semantics for APPEL which is based on ΔDSTL(x) (distributed state temporal logic [14, 15]) and then shows how this helps with the rigorous detection of conflicts.

Policy conflicts need to be first detected and then resolved. In this paper we concentrate on detecting conflicts in APPEL policies that for simplicity are assumed to be co-located (i.e. not distributed). The paper is structured as follows: In the next section we introduce the policy language and the logic we use for reasoning. In Section 3 we introduce the formal semantics for APPEL. Section 4 considers reasoning about conflict. Section 5 discusses the results and achievements. After highlighting related work in Section 6, the paper is rounded up with a brief summary and pointers to further work.

2 Background

2.1 APPEL

Policies have been used for some time to adapt the behaviour of systems at runtime. 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 [19, 22] has been developed in the context of telecommunication systems, to express end-user policies. A detailed discussion why this language was required can be found in [17].

APPEL is a general language for expressing policies in a variety of application domains with a clear separation between the core language and its specialisation for concrete domains (e.g. telecommunications). Here we concentrate on the core language; the semantics developed later maintains the separation between core and application domain. As APPEL is designed for end users rather than administrators the style of APPEL is closer to natural language allowing policies to be more readily formulated and understood by ordinary users. To aid this, a wizard has been presented to allow users to formulate policies [22].

So far, there has not been a formal semantics for APPEL – one aspect that this work is addressing. Let us consider the following syntax:

policy	::= pol_rule_group | pol_rule_group policy
pol_rule_group	::= polrule | pol_rule_group op pol_rule_group
op	::= **g**(conditions) | **u** | **par** | **seq**
polrule	::= [triggers] [conditions] actions
triggers	::= trigger | triggers **or** triggers
conditions	::= condition | **not** conditions | conditions **or** conditions | conditions **and** conditions
actions	::= action | actions actionop actions
actionop	::= **and** | **or** | **andthen** | **orelse**

Trigger and action are domain specific atoms. Condition is either a domain specific or a more generic (e.g. time) predicate.

A policy consists of a number of policy rules. The applicability of a rule depends on whether its trigger has occurred and whether its conditions are satisfied. Policy rules may be grouped using a number of operators (**seq**uential, **par**allel, **g**uarded and **u**nguarded choice) – we will discuss details when formalising their semantics.

A policy rule consists of an optional trigger, an optional condition, and an action. The core language defines the structure but not the details of these, these are defined in specific application domains. This allows the core language to be used for different purposes.

Triggers are caused by external events. Triggers may be combined using and and or, with the obvious meaning that both or either must occur. 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 have been defined to create composite actions (again, we discuss details when considering the formal semantics).

2.2 A fragment of the Distributed States Temporal Logic

ΔDSTL(x) is an extension of temporal logic to deal with global applications: it includes modalities to localize properties to system components, an operator to deal with events, and temporal modalities à la Unity [14, 15]. For instance, one may say that event Δq (q becomes true) occurring in component m when property p holds, entails that properties q and r hold in future states of components n and m, respectively: $\mathsf{m}\,(p \wedge \Delta q)$ LEADS_TO $\mathsf{n}\,r \wedge \mathsf{m}\,s$.

In this paper we need the following fragment of the logic:

$$\begin{aligned} F &::= A \mid \mathit{false} \mid \Delta A \mid \sim F \mid F \wedge F' \mid \mathsf{m}F \\ \phi &::= \exists_{\bar{x}} F \mid F \text{ LEADS\_TO } F' \mid F \text{ BECAUSE } F' \end{aligned}$$

where: A is an atom, ΔA is an event, and $\mathsf{m}F$ is a located formula[3]. A formula ϕ can be an invariant, a formula constraining the future, or a formula constraining the past: operator LEADS_TO expresses a liveness condition, F is eventually

[3] Here we consider only one component, m. The spatial features will be helpful when considering "full APPEL" that distributes policies to several sites.

followed by F'; BECAUSE expresses a safety condition, and says that F has been preceded by F'.

For the sake of readability, we leave universal quantification implicit, and make explicit existential quantifiers, when needed, i.e. in the case of invariants $\exists_{\bar{x}} F$. The intended meaning of a temporal formula is that a formula is universally quantified over all values of the variables appearing in its premises, and existentially quantified on the remaining variables.

We now show the semantics. Let $\mathcal{C}$ be a computation, i.e. a sequence of states. Let S be the set of $\mathcal{C}$'s states: s, s' are states in S and ds, ds' are distributed states in 2^S [4]. Moreover, let S be totally ordered by $\geq$, the reflexive and transitive closure of the *next* state relation. These relations are extended as follows to $2^S \times 2^S$: *ds follows (precedes)* ds' iff for each $s \in ds$ there exists $s' \in ds'$ with $s \geq s'\,(\leq)$, and for each $s' \in ds'$ there exists $s \in ds$ with $s' \leq s\,(\geq)$; *ds i–precedes* (immediately precedes) ds' iff for each $s \in ds$ there exists $s' \in ds'$ with $next(s, s')$ and for each $s' \in ds'$ there exists $s \in ds$ with $next(s, s')$. Let $\vartheta_{\bar{x}}$ be a grounding substitution for the (tuple of) variables $\bar{x}$, ϑ_F for the variables in F, and $F\vartheta$ the application of substitution ϑ to F. We say:

$$
\begin{aligned}
\mathcal{C} \models \exists_{\bar{x}} F \;&\text{ iff }\; \forall \vartheta_{F\setminus\bar{x}} \text{ each } ds \models F\vartheta_{F\setminus\bar{x}}\vartheta_{\bar{x}} \text{ for some } \vartheta_{\bar{x}} \\
\mathcal{C} \models F \text{ LEADS\_TO } G \;&\text{ iff }\; \forall \vartheta_F \text{each } ds \models F\vartheta_F \text{ is followed by a } ds' \models G\vartheta_F\vartheta_{G\setminus F} \\
&\qquad\qquad \text{for some } \vartheta_{G\setminus F} \\
\mathcal{C} \models F \text{ BECAUSE } G \;&\text{ iff }\; \forall \vartheta_F \text{each } ds \models F\vartheta_F \text{ is preceded by a } ds' \models G\vartheta_F\vartheta_{G\setminus F} \\
&\qquad\qquad \text{for some } \vartheta_{G\setminus F} \\
ds \models A \;&\text{ iff }\; \text{each } s \in ds \models A \\
ds \not\models false & \\
ds \models\sim F \;&\text{ iff }\; ds \not\models F \\
ds \models F \wedge F' \;&\text{ iff }\; ds \models F \text{ and } ds \models F' \\
ds \models \Delta A \;&\text{ iff }\; ds \models A \text{ and for } ds' \text{ i–preceding } ds,\ ds' \models\sim A \\
ds \models \mathsf{m}F \;&\text{ iff }\; \text{there exists } s \in ds \text{ such that } \{s\} \models F
\end{aligned}
$$

For instance, the following computation fragment satisfies p LEADS_TO q:

$$\rightarrow . \rightarrow p \rightarrow . \rightarrow . \rightarrow q \rightarrow . \rightarrow p \rightarrow q \rightarrow . \rightarrow . \rightarrow . \rightarrow p \wedge q \rightarrow .$$

and $\rightarrow . \rightarrow p{\wedge}q \rightarrow . \rightarrow . \rightarrow r{\wedge}s \rightarrow . \rightarrow .$ satisfies both $\mathsf{m}(p{\wedge}q)$ LEADS_TO $\mathsf{m}(r{\wedge}s)$ and $\mathsf{m}p \wedge \mathsf{m}q$ LEADS_TO $\mathsf{m}r \wedge \mathsf{m}s$. Only the latter formula is also satisfied by any of the following:

$$
\begin{aligned}
&\rightarrow . \rightarrow p \wedge q \rightarrow . \rightarrow . \rightarrow r \rightarrow s \rightarrow . \rightarrow . \\
&\rightarrow . \rightarrow p \rightarrow . \rightarrow q \rightarrow . \rightarrow . \rightarrow r \rightarrow s \rightarrow . \rightarrow . \\
&\rightarrow .p \rightarrow . \rightarrow r \rightarrow . \rightarrow q \rightarrow . \rightarrow s \rightarrow . \rightarrow .
\end{aligned}
$$

since a distributed state can be composed of distinguished and possibly non adjacent states. As a further example, the formulae $\mathsf{m}(p{\wedge} \sim p)$ and $p{\wedge} \sim p$ are

[4] In the full logic, these subset can contain states of several components, hence the name.

false, while any ds containing at least a state satisfying p and a state satisfying $\sim p$ satisfies $\mathsf{m}p \wedge \mathsf{m}(\sim p)$. On the contrary, $\mathsf{m}(p \vee q)$ is equivalent to $\mathsf{m}p \vee \mathsf{m}q$.

Some rules of the logic follows. All rules hold both for LEADS_TO and for BECAUSE: we abstract the operator by OP. Rule CC applies when formulae G and G' are located, i.e. prefixed by m, or composed of located formulae.

$$\text{CC}\frac{F \text{ OP } G \quad F' \text{ OP } G'}{F \wedge F' \text{ OP } G \wedge G'} \qquad \text{PD}\frac{F \text{ OP } G \quad F' \text{ OP } G}{F \vee F' \text{ OP } G} \qquad \text{E}\frac{F \text{ OP } false}{\sim F}$$

$$\text{SW}\frac{F' \rightarrow F \quad F \text{ OP } G \quad G \rightarrow G'}{F' \text{ OP } G'} \qquad \text{TR}\frac{F \text{ OP } G \quad G \text{ OP } H}{F \text{ OP } H} \qquad \text{I: F OP F}$$

The logic comes with MaRK, a proof assistant that partially automates the verification process and is a valuable tool supporting the proof process, making it feasible to avoid error prone "by hand" arguments [8].

3 ΔDSTL(x) Semantics for APPEL

The semantics will be developed in two steps: first of all we define rules for the interpretation of a policy rule, then we consider combining policy rules.

3.1 Semantics for a Policy Rule

Let us recall that a policy rule is essentially a triple $(triggers, conditions, actions)$, where triggers is either a single trigger or a combination of a number of them. The same holds for conditions and actions. Also recall that triggers and actions are optional.

Let us define functions $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{T}$, which will map (elements of) a policy rule into a set of ΔDSTL(x) formulae in 2^{ϕ}.

What is a trigger? Assume $t \in trigger$ and $ts \in triggers$.

$\mathcal{T}[\![\epsilon]\!] = true$

$\mathcal{T}[\![t]\!] = \Delta t$

$\mathcal{T}[\![ts \textbf{ or } ts']\!] = \mathcal{T}[\![ts]\!] \vee \mathcal{T}[\![ts']\!]$

What is a condition? Assume $c \in condition$ and $cs \in conditions$.

$\mathcal{C}[\![\epsilon]\!] = true$

$\mathcal{C}[\![c]\!] = c$

$\mathcal{C}[\![cs \textbf{ or } cs']\!] = \mathcal{C}[\![cs]\!] \vee \mathcal{C}[\![cs']\!]$

$\mathcal{C}[\![cs \textbf{ and } cs']\!] = \mathcal{C}[\![cs]\!] \wedge \mathcal{C}[\![cs']\!]$

More complicated, what is the meaning of an action? And furthermore what does it mean to compose actions? We note that actions can succeed and fail, which is important in the context of composing operations. Of course what exactly it means for an action to succeed or fail is dependent on the domain and specifics of the operation. As we are considering the semantics for the core language, we strive to stay clear of the domain specifics here. In order to capture the difference of success and failure we define two functions:

$$\mathcal{S}, \mathcal{F} : actions \rightarrow \phi \times 2^{\phi}$$

The first element in the resulting pair is a formula describing the success or failure of the action, the second element is a (possibly empty) set of side conditions that are imposing further restrictions on the first element. These extra formulae capture the rather intricate dependencies of executing an action depending on success/failure of a previous one that arise with some of the operators. Hence, for a simple action $a \in \mathit{action}$ we gain:

$$\mathcal{S}[\![a]\!] = \langle \mathsf{m}s(a), \emptyset \rangle \text{ and } \mathcal{F}[\![a]\!] = \langle \mathsf{m}f(a), \emptyset \rangle$$

Irrespective of the domain, it seems sensible to expect that an action either succeeds or fails, but never does both: $s(a) \oplus f(a)$. Let us postpone discussion of the details of $\mathcal{S}$ and $\mathcal{F}$ for a moment. In the following, assume $a \in \mathit{action}$ and $as \in \mathit{actions}$. We now have all the parts to define the meaning of a policy rule as a function $\mathcal{M} : \mathit{triggers} \times \mathit{conditions} \times \mathit{actions} \to 2^{\phi}$. Let $\mathcal{S}[\![as]\!] = \langle h_{sa}, sc_{sa} \rangle$ and $\mathcal{F}[\![as]\!] = \langle h_{fa}, sc_{fa} \rangle$, then:

$$\mathcal{M}[\![ts\ cs\ as]\!] = \{\mathsf{m}(\mathcal{T}[\![ts]\!] \wedge \mathcal{C}[\![cs]\!]) \text{ LEADS\_TO } h_{sa} \vee h_{fa}\} \cup \{sc_{sa}\} \cup \{sc_{fa}\}$$

The informal semantics for the action operators is as follows [19]:

and: This specifies that the policy should lead to the execution of both actions in either order. This can be implemented by executing the actions in a specific order or in parallel.

andthen: This is a stronger version of **and**, since the first action must precede the second in any execution.

or: This specifies that either one of the actions should be taken.

orelse: This is the **or** operator with a prescribed order. It means that a user feels more strongly about the first action specified.

Let $\mathcal{S}[\![As]\!] = \langle h_{sa}, sc_{sa} \rangle$, $\mathcal{S}[\![Bs]\!] = \langle h_{sb}, sc_{sb} \rangle$, $\mathcal{F}[\![As]\!] = \langle h_{fa}, sc_{fa} \rangle$, $\mathcal{F}[\![Bs]\!] = \langle h_{fb}, sc_{fb} \rangle$, then

$\mathcal{S}[\![As \textbf{ and } Bs]\!] = \langle h_{sa} \wedge h_{sb}, sc_{sa} \cup sc_{sb} \rangle$
$\mathcal{S}[\![As \textbf{ or } Bs]\!] = \langle h_{sa} \vee h_{sb}, sc_{sa} \cup sc_{sb} \rangle$
$\mathcal{S}[\![As \textbf{ andthen } Bs]\!] = \langle h_{sa} \wedge h_{sb}, h_{sb} \text{ BECAUSE } h_{sa} \cup sc_{sa} \cup sc_{sb} \rangle$
$\mathcal{S}[\![As \textbf{ orelse } Bs]\!] = \langle h_{sa} \vee h_{sb}, h_{sb} \text{ BECAUSE } h_{fa} \cup sc_{sa} \cup sc_{sb} \cup sc_{fa} \rangle$

and

$\mathcal{F}[\![As \textbf{ and } Bs]\!] = \langle h_{fa} \vee h_{fb}, sc_{fa} \cup sc_{fb} \rangle$
$\mathcal{F}[\![As \textbf{ or } Bs]\!] = \langle h_{fa} \wedge h_{fb}, sc_{fa} \cup sc_{fb} \rangle$
$\mathcal{F}[\![As \textbf{ andthen } Bs]\!] = \langle h_{fa} \vee h_{fb}, h_{fb} \text{ BECAUSE } h_{sa} \cup sc_{fa} \cup sc_{fb} \cup sc_{sa} \rangle$
$\mathcal{F}[\![As \textbf{ orelse } Bs]\!] = \langle h_{fa} \wedge h_{fb}, h_{fb} \text{ BECAUSE } h_{fa} \cup sc_{fa} \cup sc_{fb} \rangle$

Let us consider an example, $\mathcal{S}[\![(a \textbf{ orelse } b) \textbf{ orelse } c]\!]$, with $a, b, c \in \mathit{action}$.

$\mathcal{S}[\![(a \textbf{ orelse } b) \textbf{ orelse } c]\!]$

$\mathcal{S}[\![a \textbf{ orelse } b]\!]$

$\mathcal{S}[\![a]\!] = \langle \mathsf{m}s(a), \emptyset \rangle$
$\mathcal{S}[\![b]\!] = \langle \mathsf{m}s(b), \emptyset \rangle$
$\mathcal{F}[\![a]\!] = \langle \mathsf{m}f(a), \emptyset \rangle$

$= \langle \mathsf{m}s(a) \vee \mathsf{m}s(b), \mathsf{m}s(b) \text{ BECAUSE } \mathsf{m}f(a) \rangle$

$\mathcal{S}[\![c]\!] = \langle s(c), \emptyset \rangle$
$\mathcal{F}[\![a \textbf{ orelse } b]\!]$
$\mathcal{F}[\![a]\!] = \langle \mathsf{m}f(a), \emptyset \rangle$
$\mathcal{F}[\![b]\!] = \langle \mathsf{m}f(b), \emptyset \rangle$
$= \langle \mathsf{m}f(a) \wedge \mathsf{m}f(b), \mathsf{m}f(b) \text{ BECAUSE } \mathsf{m}f(a) \rangle$
$= \langle \mathsf{m}s(a) \vee \mathsf{m}s(b) \vee \mathsf{m}s(c),$
$\{\mathsf{m}s(b) \text{ BECAUSE } \mathsf{m}f(a), \mathsf{m}s(c) \text{ BECAUSE } \mathsf{m}f(a) \wedge \mathsf{m}f(b), \mathsf{m}f(b) \text{ BECAUSE } \mathsf{m}f(a)\} \rangle$

3.2 Semantics for a Policy Rule Group

A policy rule group is the composition of a number of policy rules. The APPEL language provides a number of operators to compose policy rules with the following informal semantics [19]:

g(condition): When two policy rules are joined by the *guarded choice* operator, the execution engine will first evaluate the nested condition. If the guard evaluates to true, the first of the two rules will be applied, otherwise the second. Clearly once the guard has been evaluated it is necessary to decide whether the individual rule is applicable. Once a guarded choice has been made, it is not undone even if the resulting rule is not followed.

u: *Unguarded choice* provides more flexibility, as both parts will be tested for applicability. If only one of the two policy rules is applicable, this will be chosen. If both are applicable, the system can choose one at random.

seq: *Sequential composition* allows the rules to be enforced in the specified order. That is we traverse the structure, determining whether the first rule is applicable. If so we apply the first rule, otherwise we check the second rule. Note that the second rule will only be checked if the first rule is not applicable.

par: *Parallel composition* of two rules allows for a user to express an indifference with respect to the order of two rules. Both rules are applied, but the order in which this is done is not important.

To define function $\mathcal{G}$, giving semantics to a policy group, we need two auxiliary functions. The first one expresses the weakest precondition for a policy rule group to be applicable. Let $(t, c, a) \in polrule$ and $ps \in pol\_rule\_group$:

$\mathcal{WP}[\![(t, c, a)]\!] = c$
$\mathcal{WP}[\![ps_1 \text{ seq } ps_2]\!] = \mathcal{WP}[\![ps_1]\!] \vee \mathcal{WP}[\![ps_2]\!]$
$\mathcal{WP}[\![ps_1 \text{ par } ps_2]\!] = \mathcal{WP}[\![ps_1]\!] \vee \mathcal{WP}[\![ps_2]\!]$
$\mathcal{WP}[\![ps_1 \text{ g(c) } ps_2]\!] = (c \wedge \mathcal{WP}[\![ps_1]\!]) \vee (\sim c \wedge \mathcal{WP}[\![ps_2]\!])$
$\mathcal{WP}[\![ps_1 \text{ u } ps_2]\!] = \mathcal{WP}[\![ps_1]\!] \vee \mathcal{WP}[\![ps_2]\!]$

The second auxiliary function is a syntactic transformation to substitute the conditions in the policies:

$d((t, c, a), x) = (t, x, a)$
$d(ps_1 \text{ op } ps_2, x) = d(ps_1, x) \text{ op } d(ps_2, x)$

We can now define $\mathcal{G} : policy\_rule\_group \rightarrow 2^{\phi}$. Here $first$, $second$ and $either$ are fresh predicates. Predicate $pick$ is randomly set.

$\mathcal{G}[\![(t, c, a)]\!] = \mathcal{M}[\![(t, c, a)]\!]$

$$
\begin{aligned}
&\mathcal{G}[\![ps_1 \text{ seq } ps_2]\!] = \\
&\qquad \mathcal{WP}[\![ps_1]\!] \longleftrightarrow first \\
&\qquad \sim \mathcal{WP}[\![ps_1]\!] \wedge \mathcal{WP}[\![ps_2]\!] \longleftrightarrow second \\
&\qquad \mathcal{G}[\![d(ps_1, first)]\!] \\
&\qquad \mathcal{G}[\![d(ps_2, second)]\!] \\
&\mathcal{G}[\![ps_1 \text{ par } ps_2]\!] = \\
&\qquad \mathcal{G}[\![ps_1]\!] \\
&\qquad \mathcal{G}[\![ps_2]\!] \\
&\mathcal{G}[\![ps_1 \text{ g(c) } ps_2]\!] = \\
&\qquad c \wedge \mathcal{WP}[\![ps_1]\!] \longleftrightarrow first \\
&\qquad \sim c \wedge \mathcal{WP}[\![ps_2]\!] \longleftrightarrow second \\
&\qquad \mathcal{G}[\![d(ps_1, first)]\!] \\
&\qquad \mathcal{G}[\![d(ps_2, second)]\!] \\
&\mathcal{G}[\![ps_1 \text{ u } ps_2]\!] = \\
&\qquad \mathcal{WP}[\![ps_1]\!] \wedge \sim \mathcal{WP}[\![ps_2]\!] \longleftrightarrow first \\
&\qquad \mathcal{WP}[\![ps_2]\!] \wedge \sim \mathcal{WP}[\![ps_1]\!] \longleftrightarrow second \\
&\qquad \mathcal{WP}[\![ps_1]\!] \wedge \mathcal{WP}[\![ps_2]\!] \longleftrightarrow either \\
&\qquad \mathcal{G}[\![d(ps_1, first \vee (either \wedge pick))]\!] \\
&\qquad \mathcal{G}[\![d(ps_2, second \vee (either \wedge \sim pick))]\!]
\end{aligned}
$$

3.3 The else operator

The example in the following section originally made use of the **else** operator in P_2 and P_3. Since it is only syntactic sugar, we are not adding it to the language presented earlier, but rather show how it can be rewritten into the considered fragment.

The informal description of **else** is that it behaves like **or** unless it occurs at the top level. So if it occurs at the top level, *if trigger and condition then* a_1 *else* a_2 is equivalent to two rules combined with a guarded choice where the condition is acting as guard, i.e. *trigger then* a_1 *g(condition) trigger then* a_2.

If the condition is empty or **else** occurs below the top level it can simply be replaced with **or**.

3.4 A non–trivial example

We will here study Example 5.7 from [19] with the purpose of showing the formal semantics at work. The purpose of the policies is to forward an incoming call when the recipient is busy. Otherwise, if not answered within 5 seconds, the call should be forwarded in a way that depends on the caller: calls from "acme" or "tom" should be forwarded to the office. If once more unanswered, the call goes to the recipient's mobile. Any other call should be forwarded home. In any case, business calls during office hours should be logged as such, and other calls as "out of hours" calls.

The policy is expressed by the policy group P_1 seq (P_2 par P_3), where:

```
P_1 = when call
      if busy
      do forward_to(vm)
```

$P_2 = P_{2a}\ \mathrm{g}(c_2)\ P_{2b}$ with:

```
c_2  = if not caller(acme) and not caller(tom)
P_2a = when not_answered(5)
       do forward_to(home)
P_2b = when not_answered(5)
         do forward_to(office)
       orelse
         do forward_to(mobile)
```

$P_3 = P_{3a}\ \mathrm{g}(c_3)\ P_{3b}$ with:

```
c_3  = if call_type(business) and calltime(h) and inbusinesshours(h)
P_3a = when call
       do log(office_hours_call)
P_3b = when call
       do log(out_of_hours_call)
```

So, following the definitions[5] , we get
$\mathcal{G}[\![P_1\ seq\ ((P_{2a}\ g(c_2)\ P_{2b})\ par\ (P_{3a}\ g(c_3)\ P_{3b}))]\!] =$

$busy \longleftrightarrow first$
$\sim busy \longleftrightarrow second$
$\mathsf{m}(\Delta call \wedge first)$ LEADS_TO $\mathsf{m}s(forward\_to(vm)) \vee \mathsf{m}f(forward\_to(vm))$
$\sim caller(acme) \wedge \sim caller(tom) \wedge second \longleftrightarrow first'$
$(caller(acme) \vee caller(tom)) \wedge second \longleftrightarrow second'$
$\mathsf{m}(\Delta not\_answered(5) \wedge first')$ LEADS_TO $\mathsf{m}s(forward\_to(home)) \vee \mathsf{m}f(forward\_to(home))$
$\mathsf{m}(\Delta not\_answered(5) \wedge second')$ LEADS_TO
$(\mathsf{m}s(forward\_to(office)) \vee \mathsf{m}s(forward\_to(mobile)))$
$\vee (\mathsf{m}f(forward\_to(office)) \wedge \mathsf{m}f(forward\_to(mobile)))$
$\mathsf{m}s(forward\_to(mobile))$ BECAUSE $\mathsf{m}f(forward\_to(office))$
$call\_type(business) \wedge calltime(h) \wedge inbusinesshours(h) \wedge second \longleftrightarrow first''$
$\sim (call\_type(business) \wedge calltime(h) \wedge inbusinesshours(h)) \wedge second \longleftrightarrow second''$
$\mathsf{m}(\Delta call \wedge first'')$ LEADS_TO $\mathsf{m}s(log(office\_hours\_call)) \vee \mathsf{m}f(log(office\_hours\_call))$
$\mathsf{m}(\Delta call \wedge second'')$ LEADS_TO $\mathsf{m}s(log(out\_of\_hours\_call)) \vee \mathsf{m}f(log(out\_of\_hours\_call))$

Remark 1. We model the application of two or more policies as an atomic step, independently of the fact that the policies are applied concurrently or in sequence. We only distinguish between *before* and *after* their application. To this purpose, we consider all predicates to be stable, including those describing the success and failure of an action. Stability means that once a predicate becomes true it stays so, and is related to the following rule: $\dfrac{F \text{ LEADS\_TO } \mathsf{m}G \wedge \mathsf{m}G'}{F \text{ LEADS\_TO } \mathsf{m}(G \wedge G')}$
This rule holds only when G and G' are stable. Since we are only interested in detecting conflict between actions, the stability assumption is reasonable. Indeed, the execution of an action does not cancel the fact that another action has been executed. Moreover, stability does not hinder the detection of situations where an action is executed when

[5] From now on, we will represent sets of formulae as lists, without brackets.

some conflicting conditions hold in the domain. In other words, we do not need to define the semantics of the actions and look at the state transformation caused by their execution. Finally, note that stability is preserved by conjunction and disjunction.

4 Dealing with Policy Conflicts

A conflict arises when, as a result of the policy application, two actions are executed, and they are defined to be conflicting in the domain description. A conflict arises also when a state is reached where a pair of conflicting predicates hold (these can be a predicate and its negation, or predicates defined to be conflicting in the domain description).
We can distinguish two types of conflict:

- actual conflict: from the policy theory and the domain description, we derive *true* LEADS_TO *conflict*. This means that the policy as it is gives raise to a conflict.
- possible conflict: from the policy theory and the domain description, we derive *true* LEADS_TO *disjunction*, and one of the disjuncts is a conflict. A typical case is when the disjunction arises from two actions, like $\mathsf{m}((s(a) \vee f(a)) \wedge (s(b) \vee f(b)))$. Distributing, one sees immediately that the conflict $s(a) \wedge s(b)$ may be avoided because one of the actions fails.

To introduce a further kind of conflict, we look at the policies in Example 5.1 of [18]:
P1 = if user(x) and if admin(x) do allow(x)
P2 = if user(Joe) do deny(Joe)
There is also a piece of domain information: $admin(Joe)$. Also, we know from the domain description that actions allow and deny are conflicting, i.e., $s(allow(x)) \wedge s(deny(x)) \rightarrow conflict$. To detect conflicts, we first express the rules in the logic:

$$\mathcal{G}[\![P1]\!] = \mathsf{m}(user(x) \wedge admin(x)) \text{ LEADS\_TO } \mathsf{m}s(allow(x)) \vee \mathsf{m}f(allow(x))$$
$$\mathcal{G}[\![P2]\!] = \mathsf{m}\, user(Joe) \text{ LEADS\_TO } \mathsf{m}s(deny(Joe)) \vee \mathsf{m}f(deny(Joe))$$

and then we develop the following proof:

$$\dfrac{\dfrac{\mathcal{G}[\![P1]\!] \quad admin(Joe)}{\mathsf{m}\, user(Joe) \text{ LEADS\_TO } \mathsf{m}s(allow(Joe)) \vee \mathsf{m}f(allow(Joe))} \quad \mathcal{G}[\![P2]\!]}{\mathsf{m}\, user(Joe) \text{ LEADS\_TO } \mathsf{m}(s(allow(Joe)) \vee f(allow(Joe))) \wedge \mathsf{m}(s(deny(Joe)) \vee f(deny(Joe)))}$$

We cannot go further. That is, we have not actually found a possible conflict, but we have discovered a *potential* one: only if the domain description is extended to satisfy the premise, i.e. if *user(Joe)* is stated, the conflict arises.

A systematic way to find all the interesting facts that might be added to the domain description and possibly generate (potential) conflicts, is to take finite consistent subsets of the Herbrand Base ($\mathcal{HB}$) of the theory obtained from the policies and the domain description. The $\mathcal{HB}$ of a theory is the set of all ground atoms which can be constructed using the ground terms and the predicate

symbols from the language fragment used to define the theory itself[6]. Since triggers and actions are not interesting extensions of the domain description, we restrict the $\mathcal{HB}$ to the atoms built using the predicates symbols in the conditions.

In our example, we have
$\mathcal{HB} = \{admin(Joe), user(Joe)\}$
Hence we can go a step further:

$$\frac{\begin{array}{l}\mathsf{m}(user(Joe) \wedge admin(Joe)) \text{ LEADS\_TO} \\ \quad \mathsf{m}(s(allow(Joe)) \vee f(allow(Joe))) \wedge \mathsf{m}(s(deny(Joe)) \vee f(deny(Joe))) \qquad \mathcal{HB}\end{array}}{true \text{ LEADS\_TO } \mathsf{m}((s(allow(Joe)) \vee f(allow(Joe))) \wedge (s(deny(Joe)) \vee f(deny(Joe))))}$$

Distributing the conjunction we get a typical case of possible conflict. Since the conflict is derived in the theory extended with $\mathcal{HB}$, we consider it to be potential.

In the following example, we consider a slight variant on the previous:

$$\mathcal{G}[\![P3]\!] = \mathsf{m}(user(x) \wedge admin(x) \wedge daytime) \text{ LEADS\_TO } \mathsf{m}s(allow(x)) \vee \mathsf{m}f(allow(x))$$
$$\mathcal{G}[\![P4]\!] = \mathsf{m}(user(Joe) \wedge nighttime) \text{ LEADS\_TO } \mathsf{m}s(deny(Joe)) \vee \mathsf{m}f(deny(Joe))$$

and apply the same the proof pattern:

$$\frac{\begin{array}{l}user(Joe) \wedge admin(Joe) \qquad \mathcal{G}[\![P3]\!] \\ \hline \mathsf{m}\, daytime \text{ LEADS\_TO} \\ \quad \mathsf{m}\, s(allow(Joe)) \vee \mathsf{m}\, f(allow(Joe))\end{array} \qquad \frac{\begin{array}{l}user(Joe) \qquad \mathcal{G}[\![P4]\!] \\ \hline \mathsf{m}\, nighttime \text{ LEADS\_TO} \\ \quad \mathsf{m}\, s(deny(Joe)) \vee \mathsf{m}\, f(deny(Joe))\end{array}}{\begin{array}{l}\mathsf{m}\, daytime \wedge \mathsf{m}\, nighttime \text{ LEADS\_TO} \\ \quad \mathsf{m}(s(allow(Joe)) \vee f(allow(Joe))) \wedge \mathsf{m}(s(deny(Joe)) \vee f(deny(Joe)))\end{array}}$$

One could factorize m, but this is not the point. To detect a potential conflict we would need to reduce the premise to *true*, by exploiting the $\mathcal{HB}$. However, any consistent subset of $\mathcal{HB}$ contains either *daytime* or *nighttime*, but not both of them. Hence it is not possible to simplify to *true*.

5 Discussion

The above method allows to detect conflicts. However, which conflict exactly is being detected depends on the definitions of the conflict 'rules'. In particular we can distinguish between two types of conflict rules that allow to detect two distinct types of conflict: conflicts between two or more policies and conflict between a policy and the system (in the absence of other policies).

Considering the relation between feature interaction and policy conflict, we can draw parallels with features. When considering features we can also find problems when a feature interacts with the system (that is in the absence of other features) – traditionally these have been considered as bugs. Feature interaction work is always based on the assumption that the individual features on their own

[6] The formal definition of $\mathcal{HB}$, in particular for the temporal case, states a set of requirements on the form of the theory (e.g. clausal, skolemized). This form is equivalent to that of the theories obtained from APPEL policies.

(of course the base system is always present) work as expected and problems occur when more than one feature is added to the system simultaneously.
Let us consider the following example:

$[\![P_1]\!] = daytime$ LEADS_TO $s(allow)$

$[\![P_2]\!] = lunchtime$ LEADS_TO $s(blacklist)$

daytime and *lunchtime* are overlapping, that is they can both hold at the same time; *blacklist* is an action.

In the light of the previous, we could say that a policy conflict is clearly a conflict between a number of policies and the problem does not occur if only one policy is present. Let us first investigate this in more detail. To detect this type of conflict, we do not require a partial specification of the actions. It is sufficient to say that $s(a)$ and $s(b)$ lead to a conflict, as we have indeed done in the previous section.

If we consider the blacklisting example at hand, the definition of conflict here would be $s(allow) \wedge s(blacklist) \rightarrow conflict$. Adding the domain dependent information $lunchtime \rightarrow daytime$, we detect the potential conflict.

In this case we do not model the fact that an action might change the value of a predicate, say *blacklisted*; we also do not model the fact that predicates might change "miracleously" (that is by other actions in the system or spontaneously). In the light of this we can see predicates in the precondition as stable.

On the other hand, a policy interacting in an undesired way with the system (in the absence of other policies) is also an interesting case to consider. It might make less sense to speak about a bug here, after all policies are not implementations of system components, but rather high level descriptions of how the system should behave. Our method allows also to detect these, however more detail and a different definition of the conflict rules is required. The conflict rules will include a notion of state variables and the actions need to be specified somewhat. For the example on blacklisting, this means that we know that $s(blacklist) \rightarrow blacklisted$, that is the action leads to a change of the predicate. Our definition of conflict then is $blacklisted \wedge s(allow) \rightarrow conflict$. It should be obvious that this conflict exists in the absence of P_2, and indeed we can detect it.

In this latter case each action comes with a (possibly empty) list of conflicting states, while in the former each action comes with a list of conflicting actions.

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 feature interaction there is only one example for a true three-way interaction, and that is quite contrived.
In some sense this question is important, as the definition of conflict could be done considering only conflicting pairs if the same holds for policies. Note that it only influences the definition of conflict rules, and thus is more important as a design guideline.

6 Related Work

Of particular relevance is the work on policy conflict: policies may contradict since they may be set by different organisations or at different levels in the same

organisation. Surprisingly, there does not appear to have been much work on policy conflicts. [7] recognises but does not address conflicts that arise in policy-driven adaptation mechanisms. [1] aims to define hierarchical policies such that, by definition, the subordinate policies cannot conflict. Conflicts are still possible if one policy in the hierarchy is changed. The use of meta-policies (policies about policies) is proposed as a solution, e.g. in [11], where meta-policy checks are applied when policies are specified and when they are executed. Similar ideas, where predefined rules and good understanding of the domain allow resolution of conflicts, are presented in [13]. In [3], it is anticipated that authorisation policies may lead to conflict. This is resolved by providing a function to compare policies and decide which should take precedence.

Further discussion on policy conflicts exist in the area of access control policies, often using logics to model policies. A formal model that permits the enforcement of complex access policies through composition is presented in [20]. Policies are expressed as safety conditions in Interval Temporal Logic, and they can be checked at run-time by the simulation tool Tempura. A fragment of first order logic, more expressive than Datalog, is used in [10]. The restrictions are such that no conflicts can arise. The logic permits to query the policy set for permissible/prohibited actions, via a friendly interface for naive users. UCON, a recent model of usage control that extends the concepts of access control has been formalized in [23], using an extension of Lamport's Temporal Logic of Actions.

Policy have also been applied to resource management in distributed system. [6] discusses the need for both static and dynamic conflict detection and resolution, and introduces computationally feasible algorithms to this purpose. The underlying model exploits a deontic logic of permission, prohibition, and obligation, coupled with temporal classifiers that indicate the span of the mode. Our approach is more flexible in expressing policies (it is not restricted to resource management and OPI type rules) and broader in scope (the conflict detection considers conflicting actions and not conflicting permissions applied to the same action).

We have made comparisons to features and feature interaction in the discussion; 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 with respect to the base system, and also with common known features.

Unfortunately, when two or more features are added to a base system, unexpected behaviour might occur. This is caused by the 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 [4]. The literature on feature interaction is large [2, 16].

7 Conclusion and Further Work

In this paper we have presented a formal semantics for a slightly reduced subset of the APPEL policy language, which sofar benefited only from an informal semantics. We also presented a novel method to reason about policy conflict in APPEL policies based on the developed semantics.

The semantics is a temporal logic theory, and a conflict is found if we derive, from the semantics of the policies, the formula $true$ LEADS_TO $conflict$, a liveness formula stating that a conflict will surely arise.

As stated earlier, policies that are being used in software systems will be created and maintained by different parties, ranging from system administrators to lay users. Clearly this scope of authors and their respective interest means that inevitably policies will conflict with each other. An automatisation of our approach, using the proof assistant MaRK [8], will lead to tool support for detecting conflicts when policies are created or changed. Note that, due to the basic structure of APPEL terms, the size of the $\mathcal{HB}$ is not an issue.

To prove the absence of a conflict, we need to derive $conflict$ BECAUSE $false$, which is a safety formula. To do so, we have to augment the semantic translation with safety conditions. This is left to further investigation.

As APPEL policies can be distributed in the networked system, we will enhance our conflict detection technique to deal with the distributed situation. ΔDSTL(x) lends itself naturally to this as the logic as concepts of location. For this it will be required to model the location information provided in policies in the logic. A further aspect is the enhancement of the formal semantics to include APPEL's user preferences.

Acknowledgements

This work has been conducted while Stephan Reiff-Marganiec was on leave at the University of Pisa, supported by the Royal Society International Outgoing Short Visit – 2006/R2 programme. The authors are all partially supported by the EU project SENSORIA IST-2005-16004.

References

1. M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In *[5]*, pages 94–112, May 2000.
2. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press (Amsterdam), 2003.
3. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *[12]*, pages 116–127, 2002.
4. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41:115–141, 2001.
5. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), May 2000.
6. N. Dunlop, J. Indulska, and K. Raymond. Methods for conflict resolution in policy-based management systems. In *Enterprise Distributed Object Computing Conference*, pages 15–26. IEEE Computer Society, 2002.

7. C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *[12]*, pages 13–24, 2002.
8. G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mark, a reasoning kit for mobility. *Automated Software Engineering*, 9(2):137–150, Apr 2002.
9. S. Gorton and S. Reiff-Marganiec. Policy support for business-oriented web service management. *la-web*, 0:199–202, 2006.
10. J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. *csfw*, 00:187, 2003.
11. E. Lupu and M. Sloman. Conflicts in policy based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6), November/December 1999.
12. J. B. Michael, J. Lobo, and N. Dulay, editors. *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, Los Alamitos, California, USA, June 2002.
13. J. D. Moffett and M. S. Sloman. Policy conflict analysis in distributed systems management. *Journal of Organizational Computing*, 4(1):1–22, 1994.
14. C. Montangero and L. Semini. Distributed states logic. In *9^{th} International Symposium on Temporal Representation and Reasoning (TIME'02)*, Manchester, UK, July 2002. IEEE CS Press.
15. C. Montangero, L. Semini, and S. Semprini. Logic Based Coordination for Event–Driven Self–Healing Distributed Systems. In R.De Nicola, G.Ferrari, and G. Meredith, editors, *Proc. 6th Int. Conf. on Coordination Models and Languages, COORDINATION'04*, volume 2949 of *LNCS*, pages 248–262, Pisa, Italy, Feb. 2004. Springer-Verlag.
16. S. Reiff-Marganiec and M. Ryan, editors. *Feature Interactions in Telecommunications and Software Systems VIII*. IOS Press (Amsterdam), june 2005.
17. S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communication services. In D. A. Peled and M. Y. Vardi, editors, *LNCS2529: Formal Techniques for Networked and Distributed Systems - FORTE2002*. Springer Verlag (Berlin), November 2002.
18. S. Reiff-Marganiec and K. J. Turner. Feature interaction in policies. *Comput. Networks*, 45(5):569–584, 2004.
19. S. Reiff-Marganiec, K.J. Turner, and L. Blair. Appel: The accent project policy environment/language. Technical Report TR-161, University of Stirling, December 2005.
20. F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *FMSE '03: Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, pages 32–42, New York, NY, USA, 2003. ACM Press.
21. M. Sloman, J. Lobo, and E. C. Lupu, editors. *Proc. 2nd International Workshop on Policies for Distributed Systems and Networks, LNCS 1995*. Springer Verlag, 2001.
22. 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.
23. 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.