Logic-based conflict detection for distributed policies

Carlo Montangero  
*Dip. di Informatica, Università di Pisa*  
monta@di.unipi.it

Stephan Reiff-Marganiec  
*Dept. of Computer Science, University of Leicester*  
srm13@le.ac.uk

Laura Semini  
*Dip. di Informatica, Università di Pisa*  
semini@di.unipi.it

**Abstract.** Policies are used to describe rules that are employed to modify (often distributed) system behaviour at runtime. Typically policies are created by many different people and there are many policies leading naturally to inconsistency between the policies, a problem that has been recognised and termed *policy conflict*. We present a novel formal semantics for distributed policies expressed in the APPEL language (so far APPEL only had an informal semantics and a recently defined formal semantics without distribution of policies). The semantics is expressed in ΔDSTL(x), 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 semantics based techniques to detect policy conflict and a consideration of conflict resolution.

1. **Motivation**

Policies have been defined as *information which can be used to modify the behaviour of a system* [24]. They are high-level statements to support personal, organisational or system goals. For instance, policies have been used in telecommunications to allow the user to express her preferences as to what to do when a call is not answered. Policies have been studied in applications such as distributed systems management, network management, Quality of Service, and access control [25, 35]. More recently, policies have been studied as means for end-users to influence the behaviour of a system. This work has mostly concentrated on telecommunications systems [36], but there have also been applications to service oriented systems [18]. Notably these approaches use a general purpose policy language, whose semantics is essentially defined by the implementation of the policy engine.

To put the use of policies in perspective, we need to look at the general run–time architecture of a policy–driven system, as shown in Figure 1. Along its execution, the System sends information on its
state to the PolicyEngine, which, in turn, reacts sending back information that influences the System behaviour. The PolicyEngine is governed by the policies stored in the PolicyBase. What makes policy-driven systems appealing, is that on the one hand policies provide a high level of abstraction and that on the other hand the PolicyBase can be updated by the user without interrupting the System operation, thus providing the user with great adaptability at a manageable and understandable level of abstraction.

In general policies are not singular entities, but are 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 [33] 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 more difficult to be modelled,
- there will be many more end-user policies than there are system policies (sheer number of policies),
- end-users are not necessarily aware of the wider consequences of a policy that they formulate.

Since conflicts hinder maximum gain when using policies, it is important to tackle the policy conflict problem. Policy conflicts need to be first detected and then resolved. In this paper, we concentrate on detecting conflicts, but also provide some insight into their resolution.

To provide the User with confidence that the rules are conflict free, it is sufficient to filter his/her input to detect those policies that, if entered in the PolicyEngine, would originate conflicts. We take a logic-based approach to this end: conflicts are detected by deducing specific formulae in a suitable theory. More specifically, the filter maintains a logical theory representing

1. the relevant information on the domain, that is, interesting facts and inference rules valid in the application domain,
2. the set of policies currently installed, i.e., contained in the PolicyBase,
3. a representation of the state space of the System, restricted to the part accessed when selecting the policies, and
4. the definition of what constitutes a conflict.
The filter also contains a deduction engine for the logic in use. Before a new policy is added to the PolicyBase, its logical representation is added to the filter theory, and then the deduction engine is run: if one of the formulae identifying a conflict is derived, the user has to deal with the detected conflict.

In [27], we have instantiated the approach using APPEL [31, 36] as policy language and engine, and the distributed-state temporal logic ΔDSTL(x) [28, 29]) to detect conflicts. A translation function has been defined in [27] to derive the ΔDSTL(x) representation of APPEL policies: as a side effect, this function defines a formal semantics for APPEL, which before was only defined informally, like most of the policy languages – as we noted above –. This paper extends the semantics to the case of distributed policies: we consider the situation in which a policy, rather than applying to the System as a whole, can be applied only to part of it. The need for such an extension arises when the System runs on physically distributed machines, and it is necessary/convenient to differentiate the policies for the various machines. Additionally, it is often convenient/convenient to distinguish the roles of the humans interacting with the System as parts to which different policies apply\(^1\). As a consequence, the locality modality of ΔDSTL(x) is used here to model both physical and logical distribution, as better suits the domain of the application at hand. This justifies the decision to use ΔDSTL(x) to give semantics to APPEL, which was made in [27].

The core concept of location is borrowed from the works on mobility, where locations permit to model several crucial issues of network–aware programming [20]. For instance, a location can represent the physical position of a mobile device, an administrative domain where components are under the control of a specific authority, a naming domain where components adopt a given naming policy, and also an address space.

The paper is structured as follows: In the next two sections we review APPEL and ΔDSTL(x) . In Section 4 we introduce the formal semantics for APPEL. Section 5 considers conflict detection and resolution. Section 6 discusses the results and achievements. After highlighting related work in Section 7, the paper is rounded up with a brief summary and pointers to further work.

2. Background: 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 [31, 36] has been developed in the context of telecommunication systems, to express end-user policies. A detailed presentation of this language can be found in [32].

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 [36].

APPEL only used to have an English language semantics, suitable for users but not for any form of formal reasoning. In [27] we have started to address this issue by providing a formal semantics for

\(^1\)More precisely, the policies apply to the software image of the human roles within the System.
a subset of APPEL. Here, we extend this semantics by including localisation of policies, something that Appel naturally provides through its `appliesTo` notation. The underlying distribution model is very simple: we just assume a flat domain of locations. The interpretation of these locations depends on the domain, as discussed in the introduction.

Let us consider the syntax given in Table 1. 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 (sequential, parallel, guarded and unguarded 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. Trigger and action are domain specific atoms. 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.

Triggers are caused by external events. Triggers may be combined using `or`, with the obvious meaning that 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).

Locations are optional, and can be used to limit the relevance of triggers and conditions so that they are considered only if they occur in specific parts of the system; similarly action can be limited to affect only parts of the system. If location is missing, these limitations are absent, so that, for instance, a condition is relevant, independently of where it is established in the running system.

It is possible to specify both a ‘policy’ location and locations for individual parts: then, the location of the part overwrites the policy location (so there will only be one location applying to each trigger, condition or action).

Locations can be expressed either as constants or variables (conventionally, constants start in lower case, and variables in upper case). Constant locations, just as triggers, conditions, and actions, depend on the domain. Variables are declared in the `appliesTo` and `when` clauses and are local to the policy. They are bound when the event triggers the policy. These values are then passed down as arguments to the conditions and actions.
C. Montangero, S. Reiff-Marganiec, L. Semini / Logic-based conflict detection for distributed policies

P1 = appliesTo ireland
when relWedding(B,G)
if over18(B,G) or courtOrder(B,G)
do registerRelWedding(B,G,ireland)
and registerLegWedding(B,G)

P2 = appliesTo Country
when relWedding(B,G)
if not over18(B,G) and not ireland: courtOrder(B,G)
do ireland: prosecute(B,G)

P3 = appliesTo Country
when relWedding(B,G)
if Country≠ireland and (over18(B,G) or ireland: courtOrder(B,G))
do ireland: registerRelWedding(B,G,Country)

Table 2. Policies for Irish marriages

As mentioned APPEL allows for policies to be located, however that notion in APPEL is restricted to a whole policy and there is no provision for locating individual parts of a policy. We have extended this slightly by allowing policy rules, triggers, actions and conditions to be located individually – clearly this extension is conservative by including the current mechanism in APPEL, but provides extra flexibility that is useful for certain application areas (e.g. Service oriented computing where a condition might depend on some other system component). As an example of how to exploit the new flexibility, consider the following scenario [1]:

If you or your partner are an Irish citizen and are thinking of getting married outside of Ireland, you should realise that the legal validity of your marriage is governed, in part, by the laws of the country in which you marry. In most, if not all cases, the legal formalities abroad are very different to those in Ireland. For example, a church marriage abroad is usually a purely religious ceremony with no legal effect. Because it is not recognised in law in the country in which it takes place, it cannot be regarded as a legal marriage in Ireland. This is the case even though a marriage in the same church or denomination in Ireland can be legally binding.

Although you must meet the foreign requirements for formalities, you are still bound by Irish law as far as the capacity to marry is concerned. For example, your marriage abroad will not be recognised under Irish law if one or both of you was ordinarily resident in Ireland and one or both of you was under 18 at the time of the marriage and did not have a Court Exemption Order.

The policies in Table 2 capture the requirements about Irish marriages. Note that, for clarity, we use infix notation for inequality. The first policy states that whenever there is a marriage in an Irish church, there will be both a religious and legal registration, provided that both the bride B and the groom G are over
eighteen years old, or that there is a court order exempting them from abiding by these requirements. The second policy states that Ireland will take action in case of an illegal wedding, i.e., one that does not abide by the Irish marriage requirements. Note that this policy applies to any country, including Ireland. Finally, P3 states that any legal marriage, though made abroad, will be registered in Ireland, for religious purposes.

The last policy also exemplifies how variables are declared and bound: the variables introduced in the appliesTo and when clauses, Country, B, and G, are bound to the values carried by the event “religious wedding of a bride and a groom in a country” that triggers the policy. These values are then passed down as arguments to the conditions and actions clauses.

3. Background: $\Delta$DSTL(x)

This extension of temporal logic was originally defined to ease the expression of properties in a setting of growing interest, namely global computing [28, 29]. $\Delta$DSTL(x) permits to deal with events and to name system components (from now on called localities) and to causally relate properties that might hold in distinguished localities, with temporal operators à la Unity [9]. For instance, one may say that event $\Delta q$ ($q$ becomes true) occurring in location $m$ where property $p$ holds, entails that properties $r$ and $s$ hold in future states in $n$ and $m$, respectively: $m (p \land \Delta q) \leadsTo n r \land m s$.

We assume a denumerable set of location names $LN = \{m, n, m_1, m_2, \ldots\}$, and a denumerable set of variables that includes the set of the location variables $LV = \{M, N, M_1, M_2, \ldots\}$.

We introduce location modalities for each location in a system: we use the location names, with a different font. For instance, $m$ is the modality corresponding to location $m$. We let quantifiers range over modalities, and $M$, $N$, $M_1$, $M_2$, . . . are location modality variables. Semantically, quantification over modality variables is standard (see [10]).

In this paper we need the following fragment of the logic ($F$ is a state formula, $\phi$ a $\Delta$DSTL(x) formula, or simply formula):

\begin{align*}
F & ::= A \mid false \mid \Delta A \mid \sim F \mid F \land F' \mid L F \\
\phi & ::= \exists \bar{x} F \mid F \leadsTo F' \mid F \because F'
\end{align*}  

where: $A$ is an atom, i.e. an elementary predicate, $\Delta A$ is an event, $L$ is a location modality, $L F$ is a located formula, and $\bar{x}$ denotes a tuple of variables. A formula $\phi$ can be an invariant $\exists \bar{x} F$, a formula constraining the future, or a formula constraining the past: operator $\leadsTo$ expresses a liveness condition, $F$ is eventually followed by $F'$; $\because$ expresses a safety condition, and says that $F$ has been preceded by $F'$. Formulæ including $\leadsTo$ or $\because$ have a premise ($F$) and a consequence ($F'$).

For the sake of readability, we leave universal quantification implicit, and make explicit existential quantifiers, when needed, i.e. in the case of invariants $\exists \bar{x} F$. A temporal formula is universally quantified over all the variables appearing in its premises, and existentially quantified on the remaining variables. For instance, $m p(x, y) \leadsTo N q(x, z)$, is implicitly prefixed by $\forall x, \forall y, \exists N \exists z$. This means that for any state $s$ of $m$ where $p$ holds for some $x$ and $y$, there is a location where, in a state that follows $s$,

\footnote{In Unity all temporal formulæ are invariants of the computation, e.g. $F \rightarrowrightarrow G$ (where $\rightarrowrightarrow$ reads leads to) is equivalent to $(\Box F \rightarrowrightarrow \Diamond G)$.}
(m) \[ p \rightarrow q \rightarrow r \rightarrow u, z \rightarrow z \]

(n) \[ p, t \rightarrow u \rightarrow v \rightarrow p \rightarrow w, t \rightarrow w, t \]

Figure 2. An example computation used as a model for $\Delta_{DSTL}(x)$ formulae.

$q$ holds for $x$ and some $z$. Finally, again for the sake of readability, we do not permit the nesting of temporal operators.

The following sections present the semantics of the logic, first informally, and then formally: The key characteristic of the logic is a novel semantic domain: the Kripke models are built on worlds that are arbitrary sets of computation states, rather than single states or tuples of them (one for each component), as it is normally proposed. This issue is further discussed in Section 7.

3.1. The logic by examples

A computation is a graph like the one in Figure 2, which describes the computation of a system with two locations. Here, $p, q, \ldots$ are the properties holding in the states, and arrows denote either local state transitions or communications (via message passing) between components. The Kripke model of this computation will be formally defined in Section 3.2. Here it is sufficient to recall that state formulae are evaluated on distributed states, i.e. arbitrary sets of computation states (any set of states would do, including for instance the set comprising two elements from $m$ and no element from $n$). We provide the informal semantics of the logic using the computation in Figure 2:

$mF$ holds in a distributed state $ds$, if and only if this set contains a state of component $m$ satisfying $F$. Consider, in Figure 2, the set containing the first two states of $m$: it satisfies $mp, mq$, and $mp \land mq$. On the contrary, it does not satisfy $m(p \land q)$: no singleton state satisfies the conjunction. Note that, on the contrary, $m(p \lor q)$ is equivalent to $mp \lor mq$.

$\Delta A$ holds when property $A$ is true, while it was false in the immediate past, i.e. $A$ becomes true in the state. Hence, in Figure 2, $m\Delta z \rightarrow mu$.

$\exists_x F$ is true always and everywhere (we call it an invariant): all distributed states (in particular all the singletons, i.e. all the computation states) must satisfy the formula. For instance, $\exists M. M(u \rightarrow z)$ means that there exists a component, $m$ in Figure 2, where each state satisfies $u \rightarrow z$.

Before discussing the temporal operators, we need to define what it means for a distributed state $ds'$ to follow another one, say $ds$: for each $s \in ds$ there must be a path to an $s' \in ds'$, and for each $s' \in ds'$ there must be a path from an $s \in ds$. For instance, in Figure 2, the distributed states containing the minimal number of states necessary to satisfy $mu \land nt$ follow any distributed state made of any choice from the first three states of $m$ and the first four of $n$.

$F \text{ LEADS TO } F'$ means that $F$ is always followed by $F'$: each distributed state satisfying $F$ is followed by a distributed state satisfying $F'$. Operator $\text{LEADS TO}$ expresses a liveness condition, and is similar to Unity’s $\rightarrow$ (leads to).
ireland \((\Delta_{\text{relWedding}}(B,G) \land (\text{over18}(B,G) \lor \text{courtOrder}(B,G)))\)

LEADS_TO ireland (done(registerRelWedding(B,G,ireland)) \land \text{done}(\text{registerLegWedding}(B,G))

Country \((\Delta_{\text{relWedding}}(B,G) \land \sim \text{over18}(B,G)) \land \sim \text{ireland}\text{courtOrder}(B,G)\)

LEADS_TO ireland done(prosecute(B,G))

Table 3. Irish marriages in \(\Delta_{\text{DSTL}}(x)\)

The computation above satisfies \(\nu u \text{LEADS_TO} m u \land n t\): a distributed state satisfies the premise if it contains, for instance, the second state of component \(n\), where \(u\) holds, call this state \(s\) and consider the distributed state \(\{s\}\). It is immediate to find a distributed state following \(\{s\}\) and satisfying the consequence, e.g. the first pair of states where \(u\) holds in \(m\) and \(t\) holds in \(n\) (related by a communication in the figure).

We also have that \(\nu w \text{LEADS_TO} m u\) holds: the \(5^{th}\) state of \(n\), where \(w\) becomes true, is followed by a state of \(m\) where property \(u\) holds. On the contrary, \(\nu w \text{LEADS_TO} m u\) is false. Indeed, as an example model we are only considering a finite fragment of a possibly infinite computation. Without knowing how the computation continues, we reason on satisfiability as if the computation were finite or –equivalently– as if none of the predicates in the finite prefix holds in the continuation. Hence, the last formula is false.

\(F \text{ BECAUSE } F'\) says that \(F\) must be preceded by \(F'\); \text{BECAUSE} is a safety operator, used to express “only if” temporal conditions. Formally, a system satisfies \(F \text{ BECAUSE } F'\) if and only if each distributed state that satisfies \(F\) is preceded by a distributed state satisfying \(F'\).

Our example computation satisfies \(\nu w \text{ BECAUSE } \nu p\land \nu u\) but does not satisfy \(\nu w \text{ BECAUSE } \nu (p \land u)\). Consider the first state, say \(s\), of \(n\) where \(w\) holds. The distributed state \(\{s\}\) is preceded by the pair \(P\) composed of the initial and the second state of \(n\). This distributed state \(P\) satisfies the consequence of the first formula. On the contrary, to satisfy the second formula, we would need a singleton state of \(n\) satisfying both \(p\) and \(u\).

As an example of a non trivial \(\Delta_{\text{DSTL}}(x)\) theory, we anticipate what we expect to be the semantics of the policies for Irish marriages, in Table 3. For clarity sake, we again use infix notation for inequality.

3.2. Formal semantics of \(\Delta_{\text{DSTL}}(x)\)

A computation is a directed graph. The nodes are partitioned into as many sets as the number of locations. There are two kinds of arches, defining local state transitions and communication, respectively. We represent both with arrows, since they cannot be confused: the former only relate nodes within a location, and define the sequence of local states inferred by the state transitions and thus a \text{next} state relationship; the latter only relate nodes in different partitions (they make a computation with \(k\) locations a \(k\)-partite graph) and define remote communications.
\[ C \models \exists x F \text{ iff } \forall ds \forall \theta_{F \setminus x} \exists \theta_x. ds \models F \theta_{F \setminus x} \theta_x \]

\[ C \models F \text{ LEADS TO } G \text{ iff } \forall ds \forall \theta_F. ds \models F \theta_F \text{ implies } \exists ds' \exists \theta_{G \setminus F}. (ds' \text{ follows } ds \text{ and } ds' \models G \theta_F \theta_{G \setminus F}) \]

\[ C \models F \text{ BECAUSE } G \text{ iff } \forall ds \forall \theta_F. ds \models F \theta_F \text{ implies } \exists ds' \exists \theta_{G \setminus F}. (ds' \text{ precedes } ds \text{ and } ds' \models G \theta_F \theta_{G \setminus F}) \]

\[ ds \models A \text{ iff } \forall s \in ds. s \models A \]

\[ ds \not\models \text{ false} \]

\[ ds \models \neg F \text{ iff } ds \not\models F \]

\[ ds \models F \land F' \text{ iff } ds \models F \text{ and } ds \models F' \]

\[ ds \models \Delta A \text{ iff } ds \models A \text{ and for } ds' \text{ i–precedes } ds, ds' \models \neg A \]

\[ ds \models MF \text{ iff } \exists s \in ds \cap S_M. \{s\} \models F \]

Table 4. Semantics of ∆DSTL(x)

We call \( S_i \) the set of states of location \( m_i \), and \( S \) the union of all \( S_i \), i.e. the set of all the states of the computation. A distributed state is any subset \( ds \) of \( S \), i.e. any set of states, and \( ds_0 \) is the set of the initial states.

Let each \( S_i \) be totally ordered by \( \geq \), the reflexive and transitive closure of the \textit{next} state relation on \( S_i \).

Let \( ds \) and \( ds' \) be distributed states in \( 2^S \). The relation \( \geq \) is extended as follows to \( 2^S \times 2^S \):

- \( ds \text{ follows (precedes) } ds' \) if and only if 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 \( s \leq s' \) (\( \geq \));

- \( ds \text{ i–precedes } ds' \) (reads as immediately precedes) if and only if for each \( s \in ds \) there exists \( s' \in ds' \) with \( \text{next}(s, s') \) and for each \( s' \in ds' \) there exists \( s \in ds \) with \( \text{next}(s, s') \). We need this notion to define the semantics of \( \Delta \). For this purpose, we also assume a tuple of auxiliary states, immediately preceding the states in \( ds_0 \), where any predicate is false. The consequence is that in the initial state \( A \rightarrow \Delta A \).

Table 4 formally defines when a computation \( C \) satisfies a formula \( \phi \); here, \( \theta_x \) is a grounding substitution\(^3\) for the (tuple of) variables \( x \), \( \theta_F \) for the variables in \( F \), and \( F \theta \) the application of substitution \( \theta \) to \( F \). To help the intuition, remember that when considering all distributed states one considers also all singletons, i.e. all computation states. Besides, just as it happens in ANPEL, the semantics of the ground atoms is domain dependent, and will not be considered further here.

The theorems of the logic that are useful for the proofs in Sections 4 and 5 follow. They deal with modality distribution in state formulae, capturing the difference between conjunction and disjunction:

\[ m(F \land F') \rightarrow mF \land mF' \]

\[ m(F \lor F') \leftrightarrow mF \lor mF' \]

\[ mF \land m\text{true} \leftrightarrow \text{false} \]

\[ ^3\text{We recall that if } F \text{ is a formula containing } X_1, \ldots, X_n \text{ as free variables, and } t_1, \ldots, t_n \text{ are ground terms, then } \theta = \{X_1 \setminus t_1, \ldots, X_n \setminus t_n\} \text{ is a grounding substitution for } F \text{ and } F \theta \text{ is the result of replacing all instances of } X_i \text{ by } t_i. \]
Table 5 lists the relevant rules for the temporal operators. All rules hold for both \textsc{leads\_to} and \textsc{cause}: we abstract the operator by $\text{OP}$. Rule CC applies when formulae $G$ and $G'$ are located, i.e. prefixed by $m$, or composed of located formulae.

The logic is supported by MaRK, a proof assistant that partially automates the verification process and is a valuable tool in the proof process, making it feasible to avoid error prone “by hand” arguments [15].

### 3.3. Stability

We say that a state formula $F$ is stable in a location $m$, denoted $\text{STABLE} m F$, if it stays true forever in a location\(^4\), once it becomes true there. Stability permits to add a rule to the $\Delta$DSTL(x) proof system, called \textsc{confluence}, which will play an important role in the next sections. The rule permits to collect modalities over conjunctions, which is not possible in general:

\[
\text{confluence} \quad \frac{F \textsc{leads\_to} m F \land m F' \quad \text{STABLE} m F \quad \text{STABLE} m F'}{F \textsc{leads\_to} m (F \land F')} 
\]

The rule is easily proved, given the definition of stability:

$C \models \text{STABLE} m F$ iff $\forall s, s' \in ds \cap S_M$ with $\text{next}(s, s')$, $\forall \vartheta_p. \{s\} \models F \vartheta_p$ implies $\{s'\} \models F \vartheta_p$

**Proof.** For the sake of readability we provide the proof only for the ground case. The extension to the general case includes the substitutions.

$F \textsc{leads\_to} m p \land m p'$ means that $ds$ satisfying the premise is followed by $ds'$ satisfying the consequence. By definition, we have $s, s' \in ds' \cap S_M$ with $\{s\} \models p$ and $\{s'\} \models p'$. Let $s'$ follow $s$ (the same reasoning holds in the opposite case). Then, for stability, we have $\{s'\} \models p$, hence $\{s'\} \models p \land p'$. Let $ds''$ be a distributed state following $ds$ and including $s'$, we have $ds'' \models m(p \land p')$, thus concluding the proof.

Finally, note that stability is preserved by conjunction and disjunction.

### 4. $\Delta$DSTL(x) Semantics for APEL

It is natural to define the interpretation of a policy rule first, and then consider their combination.

We recall that location modalities can occur at two levels within a policy: they might be introduced for the policy as a whole (using the \texttt{appliesTo} keyword) or they can occur as parts of triggers, conditions

\(^4\)For those familiar with the Unity \texttt{UNLESS}, we say that $\text{STABLE} m p \overset{\text{df}}{=} m p \texttt{UNLESS} \bot$. 

\[
\begin{array}{ccl}
\text{CC} & \frac{F \text{OP} G \quad F' \text{OP} G'}{F \land F' \text{OP} G \land G'} \\
\text{PD} & \frac{F \text{OP} G \quad F' \text{OP} G}{F \lor F' \text{OP} G} \\
\text{E} & \frac{F \text{OP} \text{false}}{F} \\
\text{SW} & \frac{F' \rightarrow F \quad F \text{OP} G \quad G \rightarrow G'}{F' \text{OP} G'} \\
\text{TR} & \frac{F \text{OP} G \quad G \text{OP} H}{F \text{OP} H} \\
\text{I} & F \text{OP} F \\
\end{array}
\]
and actions: the location provided with the \texttt{appliesTo} keyword applies to all triggers, conditions and actions inside the policy unless the same is bound to a different location.

4.1. Semantics for a Policy Rule

The semantic function

\[
P : \text{polrule} \rightarrow 2^\phi
\]

maps a policy rule into a set of $\Delta$DSTL(x) formulae, as defined in Section 3. The definition uses some auxiliary functions. First of all,

\[
T : \text{triggers} \times \text{location} \rightarrow \phi \\
C : \text{conditions} \times \text{location} \rightarrow \phi \\
A : \text{actions} \times \text{location} \rightarrow \phi
\]

map triggers, conditions, and actions into formulae, respectively. The second argument is needed to take into account the \texttt{appliesTo} clause. Moreover,

\[
SC : \text{actions} \times \text{location} \rightarrow 2^\phi
\]

maps actions into a set of side-conditions that capture the rather intricate dependencies of action execution arising with some of the action composition operators. Finally, it is necessary to manipulate syntactically some of the intermediate formulae, which is done conveniently by two transformations \texttt{DNF}: $\phi \rightarrow \phi$ and \texttt{W}: $\phi \rightarrow \phi$. We have:

\[
P[\texttt{appliesTo } m \text{ when } ts \text{ if } cs \text{ do } as] = \{\text{W}(\text{DNF}(T[ts]m \land C[cs]m)) \text{ LEADS_TO } A[as]m \} \cup SC[as]m
\]

The clause \texttt{appliesTo } m is optional, meaning that the policy applies everywhere. In this case we add the clause \texttt{appliesTo } X, where X is not free in the rest of the policy:

\[
P[\text{when } ts \text{ if } cs \text{ do } as] = P[\texttt{appliesTo } X \text{ when } ts \text{ if } cs \text{ do } as]
\]

We now consider each auxiliary function in turn.

**Triggers.** A trigger is the event or the disjunction of the events that trigger the policy. A trigger can be empty, meaning that the policy is applied also in the absence of an explicit event. Below, an empty trigger is denoted by $\epsilon$. Assume $t \in \text{trigger}, ts, ts' \in \text{triggers},$ and $m, n \in \text{location}$:

\[
T[\epsilon]m = m\text{true} \\
T[t]m = m\Delta t \\
T[n : t]m = n\Delta t \\
T[ts \text{ or } ts']m = T[ts]m \lor T[ts']m
\]
Conditions  A condition expresses properties of the state and of the trigger parameters that must be satisfied for the policy to be applied. Conditions may be combined with and, or and not with the expected meaning. Conditions are optional; we denote with $\epsilon$ an empty condition. Assume $c \in \text{condition}$, $cs, cs' \in \text{conditions}$, and $m, n \in \text{location}$, and assume De Morgan’s law has been applied to push negation down to the atomic condition level:

$$C[\epsilon]m = m \text{true}$$
$$C[c]m = m c$$
$$C[n : c]m = n c$$
$$C[\text{not } c]m = m \sim c$$
$$C[\text{not } n : c]m = n \sim c$$
$$C[cs \text{ or } cs']m = C[cs]m \lor C[cs']m$$
$$C[cs \text{ and } cs']m = C[cs]m \land C[cs']m$$

Syntactical Transformations. DNF and $\hat{w}$ are used to re-establish the intended meaning of triggers and conditions. They are needed because otherwise the semantic functions would have been more intricate. Since the definitions given above spread the modalities down to the atoms, they contradict the expected meaning, namely that triggers and conditions are evaluated in just one state per locality. The two transformations precisely obtain this: DNF simply rewrites the formula in disjunctive normal form, and need no further explanation. $\hat{w}$ factors out the location modalities in each disjunct\(^5\). The problem is not so relevant for the actions, for reasons that will become clear below, where we deal with them.

$\hat{w}$ is naturally defined in terms of the auxiliary function $\hat{u}$ that recursively aggregates the conjuncts with the same modality (4). Recursion stops when just one modality is left (5):

$$\hat{w}(\bigvee_i F_i) = \bigvee_i \hat{u}(F_i)$$ \hspace{1cm} (3)
$$\hat{u}(F) = m \bigwedge_i F_i \land \hat{u}(G) \hspace{0.5cm} \text{if } F = \bigwedge_i mF_i \land G, \text{ where } G \text{ contains no } m$$ \hspace{1cm} (4)
$$\hat{u}(\bigwedge_i mF_i) = m \bigwedge_i F_i$$ \hspace{1cm} (5)

Example 4.1. Given policy $P = \text{appliesTo } h \text{ when a if } (b \text{ and } c) \text{ or } n:d \text{ do } e$, we have:

\(^5\)Note that factoring out the location modality in a conjunction is not a sound rule in the logic. Here, we do it to force the intended meaning. Letter $\hat{w}$ stands for wedge, i.e. it should remind that this function builds the conjunction of the atoms of the same component.
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 depends 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, and concentrate on the general properties of action success and failure.

Let \( s \) and \( f \) be predicates (atoms) with one argument. For any action \( a \), \( s(a) \) is true iff \( a \) succeeded (failed). It is natural to assume that all these atoms are stable (Section 3.3). Indeed, having executed an action (either successfully or not) is a property that cannot change. Hence, for any \( m \) and action term \( a \)

\[
\text{STABLE } m s(a), \text{STABLE } m f(a)
\]

Irrespective of the domain, it seems sensible to expect that an action either succeeds or fails, but never does both in the same state in a location: \( m \sim (s(a) \land f(a)) \). Also, it is convenient to introduce a similar predicate that does not distinguish about success and failure:

\[
done(a) \leftrightarrow s(a) \lor f(a)
\]

We introduce two more auxiliary functions to deal with action success and failure:

\[
S, F : \text{actions \times locations } \to \phi \times 2^{\phi}
\]

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 capture the rather intricate dependencies of executing an action depending on success/failure of a previous one, which arise with some of the operators. Then, \( \mathcal{A} \) and \( \mathcal{SC} \), introduced earlier, simply suitably project the results of \( S \) and \( F \) on their first (\( \downarrow_1 \)) and second (\( \downarrow_2 \)) element, respectively:

\[
\mathcal{A}[as]m = \downarrow_1 (S[as]m) \lor \downarrow_1 (F[as]m) \quad \mathcal{SC}[as]m = \downarrow_2 (S[as]m) \cup \downarrow_2 (F[as]m)
\]

We first define \( S \) and \( F \) in the case of a simple action \( a \in \text{action} \). In this case there are no side conditions. Let \( m, n \in \text{location} \) in

\[
S[a]m = \langle ms(a), \emptyset \rangle \quad F[a]m = \langle mf(a), \emptyset \rangle \\
S[n : a]m = \langle ns(a), \emptyset \rangle \quad F[n : a]m = \langle nf(a), \emptyset \rangle
\]
Example 4.1 (continued). Continuing with the policy of Example 4.1:

$$\mathcal{A}[e]h = \downarrow_1 (S[e]h) \lor \downarrow_1 (F[e]h) = hs(e) \lor hf(e) = h\text{ done}(e)$$
$$\mathcal{SC}[a]h = \downarrow_2 (S[a]h) \lor \downarrow_2 (F[a]h) = \emptyset \lor \emptyset = \emptyset$$

Finally, we conclude

$$\mathcal{P}[P] = \{h(\Delta a \land b \land c) \lor (h\Delta a \land n d) \text{ LEADS_TO } m \text{ done}(e)\}$$

Example 4.2. We consider the second policy in the Irish marriages examples. For brevity, we abbreviate Country with C.

$$F_1 = T[[\text{relWedding}(B,G)]C = C\Delta \text{relWedding}(B,G)$$
$$F_2 = C[[\text{not over18}(B,G) \text{ and not ireland: courtOrder}(B,G)]C = C \overline{\text{ over18}(B,G) \land \text{ireland} \overline{\text{courtOrder}(B,G)}}$$
$$F_3 = \text{DNF}(F_1 \land F_2) = C\Delta \text{relWedding}(B,G) \land C \overline{\text{ over18}(B,G) \land \text{ireland} \overline{\text{courtOrder}(B,G)}}$$

$$W(F_3) = \hat{w}(F_3) = C(\Delta \text{relWedding}(B,G) \land \overline{\text{ over18}(B,G)}) \land \text{ireland} \overline{\text{courtOrder}(B,G)}$$

It is easily seen that $W(F_3)$ above is the premise of the second formula in Table 3. Note also that in the absence of disjunctions, DNF behaves like the identity. Then,

$$S[\text{ireland: prosecute}(B,G)]C = \langle \text{ireland} s(\text{prosecute}(B,G)), \emptyset \rangle$$
$$F[\text{ireland: prosecute}(B,G)]C = \langle \text{ireland} f(\text{prosecute}(B,G)), \emptyset \rangle$$
$$A = A[\text{ireland: prosecute}(B,G)]C = \text{ireland} s(\text{prosecute}(B,G)) \lor \text{ireland} f(\text{prosecute}(B,G))$$
$$= \text{ireland done}(\text{prosecute}(B,G))$$
$$\mathcal{SC}[\text{ireland: prosecute}(B,G)]C = \emptyset$$

Again, $A$ is the consequence of the second entry in Table 3.

The theory produced by the translation, once conjoined with that of the domain, can be used to detect conflicts, as discussed in the Section 5. First, we turn our attention to the action composition operators.

**Action Composition.** The informal semantics of the action operators is as follows [31]:

**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.
Recall that the semantics of actions is a pair, with the first element describing the actual outcome, while the second carries side conditions. Intuitively, when composing actions, the outcomes need to be combined, while side conditions must be accumulated.

Let
\[
\begin{align*}
S[as]m &= \langle h_{s(a)}, sc_{s(a)} \rangle, \\
S[bs]m &= \langle h_{s(b)}, sc_{s(b)} \rangle, \\
F[as]m &= \langle h_{f(a)}, sc_{f(a)} \rangle, \\
F[bs]m &= \langle h_{f(b)}, sc_{f(b)} \rangle,
\end{align*}
\]
then
\[
\begin{align*}
S[as \text{ and } bs]m &= \langle h_{s(a)} \land h_{s(b)}, sc_{s(a)} \cup sc_{s(b)} \rangle, \\
S[as \text{ or } bs]m &= \langle h_{s(a)} \lor h_{s(b)}, sc_{s(a)} \cup sc_{s(b)} \rangle, \\
S[as \text{ andthen } bs]m &= \langle h_{s(a)} \land h_{s(b)}, h_{s(b)} \text{ BECAUSE } h_{s(a)} \lor sc_{s(a)} \cup sc_{s(b)} \rangle, \\
S[as \text{ orelse } bs]m &= \langle h_{s(a)} \lor h_{s(b)}, h_{s(b)} \text{ BECAUSE } h_{s(a)} \lor sc_{s(a)} \cup sc_{s(b)} \rangle,
\end{align*}
\]
and
\[
\begin{align*}
F[as \text{ and } bs]m &= \langle h_{f(a)} \lor h_{f(b)}, sc_{f(a)} \cup sc_{f(b)} \rangle, \\
F[as \text{ or } bs]m &= \langle h_{f(a)} \land h_{f(b)}, sc_{f(a)} \cup sc_{f(b)} \rangle, \\
F[as \text{ andthen } bs]m &= \langle h_{f(a)} \lor h_{f(b)}, h_{f(b)} \text{ BECAUSE } h_{s(a)} \lor sc_{f(a)} \cup sc_{f(b)} \rangle, \\
F[as \text{ orelse } bs]m &= \langle h_{f(a)} \land h_{f(b)}, h_{f(b)} \text{ BECAUSE } h_{s(a)} \lor sc_{f(a)} \cup sc_{f(b)} \rangle.
\end{align*}
\]

**Example 4.3.** At this point it might be useful to see the above applied in an example. Consider the composition of actions in \texttt{appliesTo h do (a orelse n:b) orelse c}, we derive \(A\) and \(S\).

We start with the actions \(a, b, c\):
\[
\begin{align*}
S[a]h &= \langle h(a), \emptyset \rangle & S[n:b]h &= \langle n(b), \emptyset \rangle & S[c]h &= \langle h(c), \emptyset \rangle \\
F[a]h &= \langle h(f(a)), \emptyset \rangle & F[n:b]h &= \langle n(f(b)), \emptyset \rangle & F[c]h &= \langle h(f(c)), \emptyset \rangle,
\end{align*}
\]
then,
\[
\begin{align*}
S[a \text{ orelse } n:b]h &= \langle h(a) \lor n(b), \{n(b) \text{ BECAUSE } h(f(a))\} \rangle, \\
F[a \text{ orelse } n:b]h &= \langle h(f(a)) \land n(f(b)), \{n(f(b) \text{ BECAUSE } h(f(a))\} \rangle, \\
S[\text{a orelse n:b orelse c}]h &= \langle h(a) \lor n(b) \lor h(c), \{h(c) \text{ BECAUSE } h(f(a)) \land n(f(b)), n(b) \text{ BECAUSE } h(f(a)) \land n(f(b)) \text{ BECAUSE } h(f(a))\} \rangle, \\
F[\text{a orelse n:b orelse c}]h &= \langle h(f(a)) \land n(f(b)) \land h(f(c)), \{h(f(c)) \text{ BECAUSE } h(f(a)) \land n(f(b)) \land h(f(c)) \} \rangle,
\end{align*}
\]

**4.2. 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 [31]:
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 non–deterministically.

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 give semantics to a policy group we define $G : policy\_group \rightarrow 2^\phi$. We need two auxiliary functions. The first one expresses the weakest precondition for a policy rule group to be applicable. For the same reasons as we did it for the conditions in a policy rule, we need to factor out locations. Let FM (Factor out Modalities) be the composition of W and DNF, and $ps_1, ps_2 \in pol\_rule\_group$:

\[
WP[\text{appliesTo } m \text{ when } ts \text{ if } cs \text{ do } as] = FM(C[cs]m)
\]
\[
WP[ps_1 \text{ seq } ps_2] = FM(WP[ps_1] \lor WP[ps_2])
\]
\[
WP[ps_1 \text{ par } ps_2] = FM(WP[ps_1] \lor WP[ps_2])
\]
\[
WP[ps_1 g(cs) ps_2] = FM((cs \land WP[ps_1]) \lor (\sim cs \land WP[ps_2]))
\]
\[
WP[ps_1 u ps_2] = FM(WP[ps_1] \lor WP[ps_2])
\]

The second auxiliary function, $d$, is a syntactic transformation: the conditions in each policy are substituted with stronger ones computed by the semantics, to distribute the group control specification to the policy level.

\[
d(\text{appliesTo } m \text{ when } ts \text{ if } cs \text{ do } as, \text{ FM}(C[x]m)) = \text{appliesTo } m \text{ when } ts \text{ if } x \text{ do } as
\]
\[
d(ps_1 \text{ op } ps_2, F) = d(ps_1, F) \text{ op } d(ps_2, F)
\]

We can now define $G$. Here, first, second and either are fresh predicates, and pick is a predicate used to model non–determinism: its value can be randomly assigned to true or false.

\[
G[\text{appliesTo } m \text{ when } ts \text{ if } cs \text{ do } as] = P[\text{appliesTo } m \text{ when } ts \text{ if } cs \text{ do } as]
\]
\[
G[ps_1 \text{ seq } ps_2] =
\]
\[
WP[ps_1] \leftrightarrow \text{first}
\]
\[
FM(\sim WP[ps_1] \land WP[ps_2]) \leftrightarrow \text{second}
\]
\[
G[d(ps_1, \text{first})]
\]
\[
G[d(ps_2, \text{second})]
\]
The next example is non-trivial in that the policy, though not distributed, is quite complicated with several nested policy rules and alternative actions – however it is a realistic policy from a telecommunications setting.

**Example 4.4.** We consider Example 5.7 from [31]. The purpose of these policies is to forward an incoming call to the voice mail, when the recipient (here “mary”) 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:

\[
P = P_1 \text{ seq } (P_2 \text{ par } P_3)
\]

where:

\[
P_1 = \text{ appliesTo mary when call if busy do forward_to(voice_mail)}
\]

and

\[
P_2 = P_{2a} g(c_2) P_{2b}
\]

\[
P_3 = P_{3a} g(c_3) P_{3b}
\]

with:

\[
c_2 = \text{ mary: not caller(acme) and mary: not caller(tom)}
\]

\[
P_{2a} = \text{ appliesTo mary when not_answered(5) do forward_to(home)}
\]

\[
P_{2b} = \text{ appliesTo mary when not_answered(5) do forward_to(office)}
\]

\[
\text{ orelse do forward_to(mobile)}
\]

\[
c_3 = \text{ mary: call_type(business) and mary: calltime(h)}
\]

\[
\text{ and mary: inbusinesshours(h)}
\]

\[
P_{3a} = \text{ appliesTo mary when call do log(office_hours_call)}
\]

\[
P_{3b} = \text{ appliesTo mary when call do log(out_of_hours_call)}
\]
Calculating\(^6\), we get

\[
\mathcal{G}[P_1 \text{ seq } ((P_{2a} g(c_2) P_{2b}) \text{ par } (P_{3a} g(c_3) P_{3b}))] = \\
\text{mary } \text{ busy } \iff \text{ first} \\
\text{mary } \sim \text{ busy } \iff \text{ second} \\
\text{mary } (\Delta \text{ call } \land \text{ first}) \text{ LEADS TO mary } (\text{ forward_to(voice_mail)} \lor \text{ mary } f(\text{ forward_to(voice_mail)})) \\
\text{mary } (\sim \text{ caller(acme)} \land \sim \text{ caller(tom)}) \land \text{ second } \iff \text{ first}' \\
\text{mary } (\text{ caller(acme) } \lor \text{ caller(tom)}) \land \text{ second } \iff \text{ second}' \\
\text{mary } (\Delta \text{ not_answered(5)} \land \text{ first' }) \text{ LEADS TO mary } \text{ done( forward_to(home)) } \\
\text{mary } (\Delta \text{ not_answered(5)} \land \text{ second' }) \text{ LEADS TO } \\
(mary \text{ s( forward_to(office)) } \lor \text{ mary } s(\text{ forward_to(mobile)})) \\
\lor (mary f(\text{ forward_to(office)})) \land \text{ mary } f(\text{ forward_to(mobile)})) \\
\text{mary } s(\text{ forward_to(mobile)}) \text{ BECAUSE mary } f(\text{ forward_to(office)}) \\
\text{mary } (\text{ call_type(business) } \land \text{ calltime(h) } \land \text{ inbusinesshours(h)}) \land \text{ second } \iff \text{ first'' } \\
\text{mary } (\sim (\text{(call_type(business) } \land \text{(calltime(h) } \land \text{ inbusinesshours(h)))})) \land \text{ second } \iff \text{ second'' } \\
\text{mary } (\Delta \text{ call } \land \text{ first' '}) \text{ LEADS TO mary } \text{ done(log(office_hours_call)) } \\
\text{mary } (\Delta \text{ call } \land \text{ second' '}) \text{ LEADS TO mary } \text{ done(log(out_of_hours_call)) }
\]

5. Dealing with Policy Conflicts

In an application context we say that two or more policies conflict when they are applicable at the same time and their actions conflict. While this definition is only valid in a specific application domain as one must be aware of what conflicting actions are, the problem is inherent to policies. To ensure that policies can be applied it is required to remove conflicts. This process involves two stages: first of all one needs to identify whether conflict can occur, that is detect conflicts and then remove these, that is resolve conflicts.

5.1. Conflict Detection

A conflict arises when, as a result of the policy application, two actions which are defined to be conflicting in the domain description are executed. We introduce predicate conflict, to allow the user to define when two actions are conflicting. Additionally, conflicts arise 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 characterize three different situations and provide some examples looking at the policies in [33].

Actual conflict. From the policy theory and the domain description, we derive, for some locations \(m\) and \(n\), \(m \text{ true} \text{ LEADS TO n conflict}\). This means that the policy as it is gives raise to a conflict.

As an example, consider policy \(P_1 \text{ par } P_2\), where:

\[
P_1 = \text{ appliesTo myPC if user(x) do allow(x)} \\
P_2 = \text{ appliesTo myPC if user(Joe) do deny(Joe)}
\]

\(^6\)From now on, we will represent sets of formulae as lists, without brackets.
There is also a piece of domain information: myPC user(Joe) is an invariant. Also, we know from the domain description that actions allow and deny cannot fail and that they are conflicting, i.e., in any location

\[ s(allow(x)) \land s(deny(x)) \rightarrow \text{conflict} \]  

(7)

To detect conflicts, we first express the policy in the logic:

\[
\begin{align*}
\mathcal{G}[P1 \parallel P2] &= \mathcal{G}[P1] \mathcal{G}[P2], \\
\mathcal{G}[P1] &= \text{myPC user}(x) \text{ LEADS}_0 \text{ myPC} s(allow(x)) \\
\mathcal{G}[P2] &= \text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(deny(Joe))
\end{align*}
\]

We first derive a lemma:

\[
\begin{array}{c}
\text{myPC user}(Joe) \\
\hline
\text{myPC true LEADS}_0 \text{ myPC user}(Joe)
\end{array}
\]

(8)

Then we develop the following proof:

\[
\begin{align*}
\text{myPC user}(x) \text{ LEADS}_0 \text{ myPC} s(allow(x)) & \quad \text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(deny(Joe)) \\
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(allow(Joe)) & \quad \text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(deny(Joe)) \\
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(allow(Joe) \land s(deny(Joe))) & \\
\hline
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} s(allow(Joe) \land s(deny(Joe))) & \\
\end{align*}
\]

(7)

\[
\begin{align*}
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC conflict} & \\
\hline
\text{myPC true LEADS}_0 \text{ myPC conflict}
\end{align*}
\]

(8)

**Possible conflict.** From the policy theory and the domain description, we derive, for some locations \( m \) and \( n \), that \( m \text{ true LEADS}_0 F \lor n \text{ conflict} \). Of course we consider the interesting case, that is \( F \) is neither trivially true nor trivially false. This means that the policy as it is leads to a state where a disjunctive property holds, and one of the disjuncts is a conflict, hence it is *possible* that a conflict occurs.

A typical case is when the success of a pair of actions, say \( a \) and \( b \) has been defined as a conflict. If the action execution can fail, the effect of executing \( a \) and \( b \) is \((s(a) \lor f(a)) \land (s(b) \lor f(b))\) when \( a \) and \( b \) are in the same location. Considering the case where \( a \) and \( b \) are in distinct locations, one sees that the conflict \( s(a) \land s(b) \) may not occur.

In the example above, assume that action allow can fail, then we develop the following proof:

\[
\begin{align*}
\text{myPC user}(x) \text{ LEADS}_0 \text{ myPC} (s(allow(x)) \lor f(allow(x))) & \\
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} (s(allow(Joe)) \lor f(allow(Joe))) & \\
\hline
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC} (s(allow(Joe)) \lor f(allow(Joe)) \land s(deny(Joe))) & \\
\end{align*}
\]

(8)

\[
\begin{align*}
\text{myPC user}(Joe) \text{ LEADS}_0 \text{ myPC conflict} \lor \text{myPC} (f(allow(Joe)) \land s(deny(Joe))) & \\
\hline
\text{myPC true LEADS}_0 \text{ myPC conflict} \lor \text{myPC} (f(allow(Joe)) \land s(deny(Joe))) & \\
\end{align*}
\]

**Potential conflict.** This is the weakest concept: we detect a potential conflict when from the policy theory and the domain description, we derive that \( F \text{ LEADS}_0 G \lor m \text{ conflict} \). Again, we assume that \( F \) and \( G \) are neither trivially true nor trivially false.

To exemplify, we consider a modified version of \( P1 \):

\[
P1' = \text{appliesTo myPC if user(x) and admin(x) do allow(x)}
\]

with \( \mathcal{G}[P1'] = \text{myPC} (user(x) \land admin(x)) \text{ LEADS}_0 \text{ myPC} s(allow(x)) \lor \text{myPC} f(allow(x)) \).
We develop the following proof:

\[
\begin{align*}
\mathcal{G}[P1'] & \quad \text{myPC user}(Joe) \\
& \Rightarrow \text{myPC admin}(Joe) \text{ LEADS TO } \text{myPC s}(allow(Joe)) \lor \text{myPC f}(allow(Joe)) \\ 
\mathcal{G}[P2] & \quad \text{myPC admin}(Joe) \text{ LEADS TO } \text{myPC s}(allow(Joe)) \lor \text{f}(allow(Joe))) \land \text{myPC s}(deny(Joe))
\end{align*}
\]

We cannot go further. That is, we have not found any of the conflicts defined above, but still we have pinpointed a dangerous situation, where there is a potential for a conflict. We need to consider two cases, namely whether Joe is an administrator of myPC or of a different machine. A conflict arises only if the domain description is extended to satisfy the premise, i.e. if myPC admin(Joe) is stated. We call this a potential conflict.

To characterize the potential conflicts, we need to simulate all the dangerous evolutions of the domain and of the System state, that is, a way to find all the interesting facts that might be added to the current theory and possibly introduce conflicts. A systematic and constructive way to do this, is to take the finite consistent subsets of the finite Herbrand Base (HB) of the theory obtained from the policies and the domain description. The 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\(^7\). We restrict the HB to the atoms built using the predicates symbols in the conditions, as these provide further insight to the occurrence of conflicts.

In our example, we have

\[ HB = \{\text{admin}(Joe), \text{user}(Joe)\} \]

Hence we can go a step further in the proof above:

\[
\begin{align*}
\text{myPC admin}(Joe) \text{ LEADS TO } \text{myPC s}(allow(Joe)) \lor \text{f}(allow(Joe))) \land \text{myPC s}(deny(Joe)) & \quad \text{HB} \\
\text{myPC true LEADS TO myPC ((s(allow(Joe)) \lor f(allow(Joe))) \land s(deny(Joe)))}
\end{align*}
\]

Distributing the conjunction we get a typical case of possible conflict. Since the conflict is derived in the theory extended with HB, we consider it to be potential. Atoms in the Herbrand Base are not located. In the proofs we can weaken them from invariant to located to better understand the source of a potential conflict, in this case we locate admin(Joe).

### 5.2. Conflict Resolution

Conflict resolution can in general be attempted in a number of ways, and which is best suited depends on the situation. We can broadly distinguish between resolution at design-time and resolution at run-time. Considering the taxonomy of policy conflict [33], the domain entity is the crucial point with regard to distribution of policies. The taxonomy makes explicit that “the more entities are involved and the greater their independence, the less knowledge will be available of policies that concern them”.

We can conclude from this that design-time resolution is always feasible when policies are co-located and owned by the same user. In this case resolution will be a redesign of the policies. However, when policies are more distributed, this will become less feasible as conflicts often can only be dealt with at

---

\(^7\)The formal definition of 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 APPEND policies.
run-time – resolution will then be the violation (or non-enactment of an action) of policies with lesser
precedence. Nevertheless, there is a wide spectrum between the two extremes of co-location and com-
plete distributedness, and any conflict that is resolved before run-time is of benefit.

The default mechanism in the APPEL policy enforcement environment is that all policies that are trig-
gerated and whose conditions hold will be applied in parallel – that is there will be no semantic difference
between two independent policies ‘\(P_1\)’ and ‘\(P_2\)’ and ‘\(P_1 \text{par} P_2\)’.

As an example we consider the car repair scenario as introduced in the Sensoria project [2, 16]. Assume a car equipped with a diagnostic hardware/software system that continuously reports on the
status of the vehicle. When the car experiences some major failure (e.g. engine overheating, exhausted
battery, flat tires) an embedded service is invoked to provide the user with a tow-truck that carries his
damaged car to a garage for repair. A potential conflict can be detected in this scenario.

**P.1:** If a car fault happens, then tow-truck and garage services are automatically selected.

Policy P.1 is defined by the car manufacturer, and it is deployed in the diagnostic system. This scenario
can be modified to take care of user needs. We introduce a new policy to accommodate the fact that
usually a driver already knows and trusts some of the garage and towing truck services of his/her own
town.

**P.2:** If a car fault happens in the driver’s town, then he wants to select the services to be used.

We state these policies in APPEL:

\[
\begin{align*}
P_1 &= \text{when car\_accident do aut\_sel} \\
P_2 &= \text{appliesTo driver\_Town when car\_accident do man\_sel}
\end{align*}
\]

where we abbreviate automatically select services with aut\_sel, and similarly for the manual selection.

We derive the policies semantics. In line with the definition in section 4.1 we need to add a location
variable to the unlocated policy P.1, which for clarity we call AnyTown:

\[
G[P_1] = \text{AnyTown} \triangleleft \text{car\_accident leads\_to AnyTown done(aut\_sel)}
\]

\[
G[P_2] = \text{driver\_Town} \triangleleft \text{car\_accident leads\_to driver\_Town done(man\_sel)}
\]

It is reasonable that the selection is done either manually or automatically, i.e.:

\[
done(\text{aut\_sel}) \wedge done(\text{man\_sel}) \rightarrow conflict
\]

Since a legal instance of \(G[P_1]\) is when location driver\_Town binds the location variable AnyTown:

\[
\text{driver\_Town} \triangleleft \text{car\_accident leads\_to driver\_Town done(aut\_sel)}
\]

we can derive (in two steps) that in driver\_Town the services are selected automatically and manually at
the same time, leading to a conflict:

\[
\begin{align*}
\text{driver\_Town} \triangleleft \text{car\_accident leads\_to driver\_Town done(aut\_sel)} & G[P_2] \\
\text{driver\_Town} \triangleleft \text{car\_accident leads\_to driver\_Town (done(aut\_sel) \wedge done(man\_sel))}
\end{align*}
\]
This is a typical situation where it is not feasible to operate at design-time: the manufacturer policy is already deployed in the system, and the user one is added later on.

According to what we discussed in [17], to reduce conflicts when a new policy is introduced, the user should define its relation with other policies applying to the same system. Usually, (s)he should assign a priority to the most recently introduced policy. In APPEL, this can be done using the APPEL operator seq, which implements a priority: in \( P_1 \text{ seq } P_2 \) we determine whether the first policy is applicable, if so we apply it, otherwise we check the second one.

Back to the car repair example, we note that policy \( P_2 \) has a natural priority with respect to the default policy \( P_1 \). We thus compose them in the following sequence:

\[
P = \text{ appliesTo } \text{driverTown when car\_accident do man\_sel }
\text{ seq }
\text{ appliesTo } \text{AnyTown when car\_accident do aut\_sel}
\]

To be precise, and to understand the following, we note that in both policies there is no condition, meaning that it is true. For clarity of this explanation, we make this true condition explicit: the semantics of seq is based on the WP, which is defined in terms of conditions.

\[
G[P_1] = \text{AnyTown} (\Delta \text{car\_accident } \land \text{true}) \text{ LEADS\_TO AnyTown done(aut\_sel)}
\]

\[
G[P_2] = \text{driverTown} (\Delta \text{car\_accident } \land \text{true}) \text{ LEADS\_TO driverTown done(man\_sel)}
\]

We have \( WP[P_1] = \text{AnyTown true} \) and \( WP[P_2] = \text{driverTown true} \). Hence:

\[
G[P] = G[P_2 \text{ seq } P_1]
= \text{driverTown } \Delta \text{car\_accident LEADS\_TO driverTown done(man\_sel)}
\]

\[
\text{AnyTown } \Delta \text{car\_accident } \land \sim \text{driverTown true LEADS\_TO AnyTown done(aut\_sel)}
\]

Now, in any locality \( n \) different form \( \text{driverTown} \), the first policy does not apply, while an instance of the second formula does, namely:

\[
n \Delta \text{car\_accident } \land \sim \text{driverTown true LEADS\_TO n done(aut\_sel)}
\]

In location \( \text{driverTown} \) the first policy can apply while the second cannot. Indeed, substituting \( \text{driverTown} \) for \( \text{AnyTown} \) we get:

\[
\text{driverTown } \Delta \text{car\_accident } \land \sim \text{driverTown true LEADS\_TO driverTown done(aut\_sel)}
\]

i.e. a formula with a premise that no distributed state \( s \) can satisfy, since the first conjunct requires that \( s \) contains at least a state of \( \text{driverTown} \), while the second just tells the opposite.

6. Discussion

We address two topics that in our opinion deserve special attention: conflicts between a policy and the system, and the so-called “three-way interaction” and its influence on reasoning on the absence of conflicts.
Conflicts between a policy and the system. 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 (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:

\[
P1 = \text{if} \ \text{daytime} \ \text{do} \ \text{allow} \\
P2 = \text{if} \ \text{lunchtime} \ \text{do} \ \text{blacklist}
\]

*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 \(s(a) \land s(b)\) 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 \(s(\text{allow}) \land s(\text{blacklist}) \rightarrow \text{conflict}\). Adding the domain dependent information *lunchtime* → *daytime*, we detect the potential conflict.

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 simply means letting *blacklisted* be a possible predicate holding in the state, and \(s(\text{blacklisted}) \land s(\text{allow}) \rightarrow \text{conflict}\). Since *blacklisted* is a predicate which may hold independently of P2, a conflict may exist also in the absence of P2, 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.

In this paper we have not considered how exactly one arrives at the notion of which actions are conflicting as this is domain dependent. Nevertheless, it is of course an interesting avenue of research which complements the presented work. We made the (not unrealistic) assumption that the facts of what constitutes conflicts are provided by a domain specialist. Some initial work in the telecommunications domain [23] has shifted the domain expertise to pre- and post-conditions of actions (that is, which specific system state will exist before an action can be executed, and which changes does it apply to the state) and then used formal reasoning in the form of the Alloy model-checker to detect conflicting actions.
The “three-way interaction”. One further aspect to consider, and this is again based on experience in feature interaction, is the question as to how many policies are required to generate a conflict. In the community discussions have taken place around a topic called “three-way interaction”. In the feature interaction detection contest at FIW2000 [8] this was an issue, and the community decided that there are two types of three-way interaction: those where there is already an interaction between one or more pairs of the three features and those where the interaction only exists if the triple is present. The latter were termed “true” three-way interactions.

Nothing has been written about true three-way interaction, as only one, quite contrived, example of such an interaction has been found. We can hence consider as realistic the assumption that no “true” three-way interaction may occur.

Now, consider the issue of proving the absence of a conflict. From a formal point of view, we cannot say anything, due to the undecidability results of 1st order logic. However, from a pragmatical point of view, we can extend the APPEL semantics with some safety formulae. For instance, the semantics of

\[ P_1 = \text{appliesTo mary when call if daytime do answer} \]

would be the pair of formulae, the second being immediately derived from the first:

\[ \text{mary}(\Delta \text{call} \land \text{daytime}) \text{ LEADS_TO } \text{mary answer} \]

\[ \text{mary answer BECAUSE mary}(\Delta \text{call} \land \text{daytime}) \]

This is actually a restrictive interpretation, saying that Mary answers only after receiving a daytime call. The advantage is that we can look for those policies whose actions are conflicting with answer, for instance

\[ P_2 = \text{appliesTo mary when call if nighttime do ignore} \]

and prove the absence of conflicts by deriving conflict BECAUSE false:

\[ \text{mary answer BECAUSE mary}(\Delta \text{call} \land \text{daytime}) \text{ mary ignore BECAUSE mary}(\Delta \text{call} \land \text{nighttime}) \]

\[ \text{mary (answer \land ignore) BECAUSE mary}(\Delta \text{call} \land \text{daytime} \land \text{nighttime}) \]

\[ \text{conflict BECAUSE false} \]

The no “true” three-way interaction assumption permits to conclude that if no pair of policies are conflicting, then the whole specification is free of the addressed conflict.

7. Related Work

Defining a world to be a singleton state is inadequate when reasoning on logical relations between state formulae like the premises or the consequences of a \( \Delta \text{DSTL}(x) \) formula. For instance, \((nq \land mr) \rightarrow nq\) would permit to weaken the consequences of \(mp \text{ LEADS_TO } nq \land mr\). However it would be a useless formula, since no world can satisfy the conjunction \(nq \land mr\).

Most logics for distributed systems let worlds be tuples of states, one for each component. This choice would induce a form of synchronization and remote knowledge which is meaningless in an asynchronous setting. For instance, assume a pair of worlds \(w\) and \(w'\) with \(w'\) following \(w\), with \(p\) holding
in the n state of w and q holding in the m state of w'. Then, one can assert that np leads_to mq holds, even in the absence of any communication between these two states.

Hybrid logic allows the specifier to directly refer to specific points (states) in the model, through the use of nominals [5]. A nominal i is an atom which is true at exactly one point in any model. The operator @i permits to jump to the point named by nominal i. We might consider defining an hybrid signature including distinguished sets of state variables, one for each component, and translate m F in \( \exists x. \@x F \), where x is a state variable in the appropriate set. However a main difference persists, due to the possibility of jumping: the value formula \( \exists x. \@x F \) is independent from the current point of evaluation, while this is not the case for m F, which is always false when evaluated in a state not in m. Indeed, the \( \Delta DSTL(x) \) axiom system includes a dead–end axiom m \( \neq n \rightarrow (mnF \equiv \bot) \) to prevent making assertions on the current state of a remote component. Only future and past remote states can be accessed, since their properties can be known through some remote communication.

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. [14] recognises but does not address conflicts that arise in policy-driven adaptation mechanisms. [3] 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 [24], 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 [26]. In [6], 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 [34]. 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 [19]. 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 [37], using an extension of Lamport’s Temporal Logic of Actions.

Policies have also been applied to resource management in distributed system. [13] 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).

Additionally there is interest in policy languages to express system management policies, notably Ponder (and Ponder 2) [11], PMAC-ACPL [21] and CIM-SPL [12]. Policy conflict is an issue for all these languages and hence it is interesting to see if the presented methods can be applied in their context. We believe that this is the case, albeit there are some problems. Most notably, these languages do not possess formal semantics (Ponder does to some extent) – their semantics is defined by execution environments. ACPL and CPL are essentially event-condition-action rules with some extras, and hence are quite close to APPEL so we can assume that the presented work can be relatively easy transferred to
these. ACPL has a notion of location in form of its scope; it has no grouping of policies as far as we can tell – so it is simpler in some aspect but would definitely benefit from the reasoning about localised policies presented here. SPL has groups of policies but no locations – again the presented work should be applicable with little effort. Ponder is somewhat different in that its main policy types are obligation and authorization policies placing it into the domain of deontic logics and hence making the transferability of the presented ideas less clear.

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 a 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 commonly 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 [7]. The literature on feature interaction is large, e.g. [4, 8, 30].

According to taxonomy introduced in [22], the policies considered in the paper are action policies, i.e., policies that “dictate the action that should be taken whenever the system is in a given current state”. There are two other kind of policies, goal and utility function policies.

Goal policies are higher level than action policies: rather that explicitly specify the action to be taken in a given state, they define a set of desirable states, leaving the system to decide the actions to be taken. For instance, in the Irish marriages case, one might state, in a pseudo–language:

\[
\text{in Ireland wedding}(B,G) \text{ requires (over18}(B,G) \text{ or courtOrder}(B,G))
\]

The framework we defined in the paper can be used to formalize it. The following formula expresses an invariant (no existential quantifier here, the policy applies for all brides and grooms):

\[
\text{ireland} \Delta Wedding(B,G) \rightarrow (\text{over18}(B,G) \lor \text{courtOrder}(B,G))
\]

Utility function policies add some flexibility to goal policies: they assign a desirability value (utility) to each state. The objective is to maximize utility, and conflict detection is not an issue here. As in the goal policies case, the actions to be taken are not explicit, but left to the system. In the Irish marriages example one might define the utility of a wedding as a function of the ages of bride and groom. Then the system has to try to marry people in the best possible way. We claim that this is not an issue where the logic can help.

8. Conclusion and Further Work

In this paper we have presented a formal semantics for the APPEL policy language, which so far benefited only from an informal semantics. An initial formalisation of a subset of the language had been presented earlier [27], but this did not address the semantics for the distributed policies. We also presented a novel method to reason about policy conflict in APPEL policies based on the developed semantics and have touched on conflict resolution.
As APPEL policies can be distributed in the networked system, ΔDSTL(x) lends itself naturally to formulate the semantics as the logic permits to deal with the concept of location.

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 policy authors and their respective interest means that inevitably policies will conflict with each other, and that tool support is needed. We plan to use the proof assistant MaRK [15] to implement the conflict detection filter mentioned in the introduction. MaRK comprises an environment for reasoning in ΔDSTL(x): the user can input a ΔDSTL(x) theory, and use the proof system to prove theorems. Being built on top of Isabelle, MaRK leverages its capabilities for automating proofs: the most challenging task in implementing the conflict detection filter is how to exploit these capabilities to get completely automatic detection. Since the context of the proofs is highly constrained – the goals have a definite structure as do the axioms in the policy theory – the goal seems reasonable. The other tasks of the implementation, namely the construction of the ΔDSTL(x) theory from the APPEL policies and of the Herbrand Base HB for potential conflict detection are standard software engineering jobs. Note that, due to the basic structure of APPEL terms, the size of the HB is not an issue.

One aspect of APPEL that we left for future investigation is the possibility to express user preferences. This mechanism allows for policies to include a statement as to how strongly a user feels about them. So for example one might specify that “I must not be disturbed at dinner” or “family members might access my personal records”. These preferences are interesting from a practical point: they can help with deciding automatically on which policy to sub-ordinate when resolving conflicts at system run-time. When considering how to model their semantics one can consider them as an additional modality.

Acknowledgments

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

References


