Expressive Policy Analysis with Enhanced System Dynamicity

ABSTRACT
Although several research efforts have been devoted to the issue, the effective analysis of policy based systems remains a significant challenge. Policy analysis should at least (i) be expressive (ii) take account of obligations and authorizations, (iii) include a dynamic system model, and (iv) give useful diagnostic information. We present a logic-based policy analysis framework which satisfies these requirements, showing how many significant policy-related properties can be analysed, and we give details of a prototype implementation.

Keywords
Policies, Formal analysis, Security and protection, Authorization

1. INTRODUCTION
There is an obvious relationship between the expressiveness of a policy language and the ability to analyse its properties and impact on system behaviour. Without being expressive a policy language may not be able to regulate complex system behaviour, apply across heterogeneous components, or apply to systems involving frequent changes such as mobile systems. Yet without analysis much of the benefit of using policy-based techniques and declarative policy languages may be lost. Arguably, the lack of effective analysis tools accounts in part for the lack of wider adoption of policy-based techniques. A policy framework ought to satisfy several key properties.

First, that of expressiveness. We need to be able to specify both authorization [19, 29] and obligation policies [17, 27, 28] formally, and to do so in a way which recognizes that there is often a complex dependence of one on the other—so that, for example, granting permission to access a certain resource may depend on whether or not an obligation to present a certain credential has been satisfied. To be expressive, a framework should also allow policy decisions to depend on aspects of the evolving system history, so that authorization can depend not only on a static assignment of roles, or the fixed location of a sensor, but will vary as the system state changes. Policies must also provide fine-grained defaults. Many policy languages rely on a simple, universal default of either permitting or denying requests not covered by any specific policy rule. SELinux [23], for example, has blanket denials for actions not covered explicitly by policy rules. Whilst we can support such defaults, there is a need for a much more nuanced control over the default behaviour, so that requests to delete a file may be denied by default, but requests to read a file would be authorized [18]. Defaults are also useful in the presence of conflicts: the policy combination rules of XACML [27], for example, specify the response to a request which the explicit policy rules both authorize and deny. Our framework can represent both sorts of default easily and concisely.

Second, a powerful policy analysis component is essential. This lets the policies be checked for necessary or desirable properties. Existing analysis frameworks—such as [17,6] and [5]—rarely take into account the changing system state, or only allow the statement and analysis of temporal constraints and relationships amongst policy decisions. There is a strong relationship between the expressiveness of a policy language and its analysis, since a less expressive language simplifies the analysis but also limits the scope of properties that can be checked. For example, if the language does not allow the representation of an authorization’s dependence on the fulfilment of two key obligations, then an analysis of whether or not it was even possible to jointly satisfy those two obligations and then obtain permission, would not be possible.

Yet, expressiveness in the representation of policies, in the way they rely on each other, and the way in which they interact with the system, is only one half of the expressiveness needed for an analysis component. This is because policy authors also need to be able to check for a wide variety of properties on the policies and systems they define. This requires an expressiveness, not merely in policy and system representation, but in the query language of the analysis component, and strength in the analysis algorithm itself. Analysis should be able to cope with the following query tasks:

- **Modality conflicts** such as the joint authorization and denial of a request to perform some action, or the presence of an obligation to act without the permissions necessary for its fulfilment.
- **Separation of duty** conflicts, including static separation of duty, dynamic, and many other classes (see
We have also developed a prototype implementation,\(^1\) of our analysis framework, allowing the representation of systems descriptions and policies, and supporting the analysis tasks we discuss in this paper.

The paper is organized as follows. Section 2 considers related work. Section 3 gives the syntax and semantics of the language, including a number of illustrative examples. Section 4 discusses the kinds of analysis our language permits, together with a discussion of the implementation and complexity properties. Conclusions and directions for future research are in Section 5. Details on the the abduction and consistency algorithms are included in the appendix.

2. RELATED WORK

The Lithium language of Halpern and Weissman [16] is a logical formalism for policy representation and analysis; however, the authors work in pure first-order logic which imposes on the policy author the burden of specifying complete definitions (every request has a decision) since default decision policies are not expressible. For example, representing that all and only faculty members are permitted to chair committees; students are not [16] requires a complete specification of faculty and student body members, which may change dynamically, thus raising the well-known problem of elaboration tolerance [24]. The use of default rules—of the kind that our formalism supports—can simplify specifications and changes to the specification and provide elaboration tolerance. Another important difference in our work is that we perform hypothetical analysis through abduction, letting the engineer specify initial conditions and sequences of requests or events in a system only partially; our analysis algorithm then supplies the additional information which makes a property true or false.

Our treatment of obligations is based on our own experience with Ponder [11] and deontic logic; the result is similar to [17]. However, we have adapted obligation policies to produce a more general language that allows more complex policies to be represented, and our framework is capable of supporting analyses such as the strong accountability checking presented in [17]. Dougherty et al. [12] present a model of obligation policies in which obligations are tied to authorizations, as conditions on acquiring permission to access a given resource. The model we use is in this respect more general, allowing obligations that are not tied to authorizations, as well as mutual dependence. [12] also includes a system model, though this is conceived abstractly as a set of state traces, which would need to be defined in full—our use of Event Calculus domain descriptions allows us to generate traces of actions which lead to the holding or violation of policy-related properties, by using concise system descriptions.

Barker presents in [3] a language that supports specification of access control policies using stratified clausal-form logic, with emphasis on RBAC policies. However, this work does not discuss analysis. The Authorization Specification Language (ASL) [20], the Flexible Authorization Framework (FAF) [19] and the extension to handle dynamic authorizations discussed in [9] are also based on stratified clausal-form logic. They offer techniques for detecting modality conflicts and some application-specific conflicts in authorization poli-

\(^1\)Reference removed through anonymity requirement on submission.
cies. However, they work with a fixed domain model; our framework does not assume a predefined domain model and can cater for varying system descriptions.

In [4], a language is presented for the expression of access control policies, with an associated analysis framework based on a subset of transaction logic programs; in these respects the approach is similar to our own. However, although the authors do take into account the fact that some policy-governed actions can change role activations, and thus there is some dynamicity in their framework, they limit the specification to sequences of operations. Further, the classes of policy they express are limited: they cannot represent explicit prohibitions, and are thus forced into an unchangeable default assumption that anything not explicitly allowed is to be denied. Our formalism is more expressive: it has explicit prohibitions, and a great degree of control can be exercised in the way defaults cover policy gaps, or legislate between conflicts. In [5], the authors use abduction to analyze authorization policies, particularly focussing on finding explanations for the denial of access requests. They provide soundness, completeness and termination results. However, as with [4], there is no fully dynamic system model, so that although credentials can be abduced which would have led to the granting of access, it is not possible to see which policy-regulated actions, or system events, would have led to those credentials being present.

3. POLICIES

3.1 Preliminaries

Our operational model broadly follows the architecture and operation of XACML [27]. There is a policy component, consisting of policy decision and enforcement points (PDP/PEP), and the system to which policies refer and which they modify. The PDP has access to a policy repository. Authorization decisions are made in response to requests for a subject to perform an action on a target, using the policies, and these decisions are then enforced by the PEP. The PDP also monitors whether obligations of subjects to perform actions have been met or not. Systems move between states depending on the occurrence of actions and events—some of which may be controlled by policies, some not.

We use many-sorted first-order predicate logic as our base language, and clearly distinguish the policy representation language from the domain description language. This allows us to detach policy representations from system representations, and compare the implementation of a policy in different systems easily. The policy representation language, \( L^* \), includes sorts for the Subjects, Targets and Actions mentioned in policies, together with a sort and constants for Time, which we represent using the non-negative reals. Standard arithmetical functions (+, −, ÷, /, *) and relations (=, ≤, ≥ etc.) are assumed. The predicates of \( L^* \), which we call regulatory predicates, are shown in Table 1. The predicates permitted, denied are self-explanatory. A particular instance \( req(\text{sub}, \text{tar}, \text{act}, t) \) means that a request for \( sub \) to perform \( act \) on \( tar \) is made at time \( t \). Instances \( obl(\text{sub}, \text{tar}, \text{act}, t_1, t_2, t) \) and \( fulfilled(\text{sub}, \text{tar}, \text{act}, t_1, t_2, t) \) (or \( violated(\text{sub}, \text{tar}, \text{act}, t_1, t_2, t) \)) denote that at time \( t_1 \) sub is placed under an obligation to perform \( act \) on \( tar \) between \( t_1 \) and \( t_2 \), and that the obligation with these parameters has been fulfilled (resp. violated). Finally, a given instance

<table>
<thead>
<tr>
<th>Input regulatory</th>
<th>Output regulatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>req(Sub, Tar, Act, T)</td>
<td>do(Sub, Tar, Act, T)</td>
</tr>
<tr>
<td>deny(Sub, Tar, Act, T)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State Regulatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>permitted(Sub, Tar, Act, T)</td>
</tr>
<tr>
<td>denied(Sub, Tar, Act, T)</td>
</tr>
<tr>
<td>obl(Sub, Tar, Act, T, T, T)</td>
</tr>
<tr>
<td>fulfilled(Sub, Tar, Act, T, T, T)</td>
</tr>
<tr>
<td>violated(Sub, Tar, Act, T, T, T)</td>
</tr>
<tr>
<td>cease_obl(Sub, Tar, Act, T, T, T, T)</td>
</tr>
</tbody>
</table>

Table 1: Policy analysis language \( L^* \): the predicates

case_obl(sub, tar, act, t_in, t, t, t) is true at time \( t \), if an obligation initially contracted by sub at \( t_{in} \) to perform \( act \) on \( tar \) between \( t_s \) and \( t_e \) is no longer binding.

The domain description language \( L^D = L^D_{EC} \cup L^D_{stat} \) is used to represent both changing and unchanging properties of the system regulated by the policy. We use the Event Calculus [22] (EC) to model this dynamicity in our domains. The language includes sorts Fluent (for dynamic features of states), Event (for system events not regulated by policies), Occurrence (for representing system events regulated by policies) and Time (as before). The predicates of \( L^D_{EC} \) are standard in the EC (see Section 3.4); the predicates of \( L^D_{stat} \), for unchanging properties of systems, are user-defined.

The operational model and function of the language categories described above is illustrated in Figure 1. A complete formal definition of the language is given in [2].

3.2 Authorizations

First, a preliminary definition.

Definition 1 A time constraint \( C \) is an expression of the form \( \tau \rho \tau_2 \), where each \( \tau \) is a constant or variable of type Time, or an arithmetic linear expression built using \(+, −, ÷, /, *, \lt, \gt, \leq, \geq \).

Note that in the following definition, and in most other clauses we present in this paper, the time \( T \) in the head of the rule is a variable, rather than a fixed time—this means that the same rule can be applied whenever the conditions in the body become true.

Definition 2 An authorization rule is a formula

\[
[\text{permitted/denied}](\text{Sub}, \text{Tar}, \text{Act}, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.
\]

1. the \( L_i \) are atoms taken from the set \( \text{L}^* \cup \text{L}^D_{stat} \cup \{\text{holdsAt, happens, broken}\}^2 \) possibly preceded by the negation-by-failure \textbf{not} ; the \( C_i \) are time constraints;

2. any variable appearing in a time constraint must also appear somewhere other than in a time constraint;

\(^2\)For the meaning of these predicates, see Section 3.4.
3. Sub, Tar, Act, T are terms of type Subject, Target, Action and Time respectively.

4. for the time argument $T_i$ of each $L_i \notin L_{\text{stat}}$, we must have $C_1 \land \cdots \land C_n \models T_i \leq T$; if $C_1 \land \cdots \land C_n \models T_i = T$ then the $L_i$ must not be an output regulatory predicate and if in addition $L_i \in L_{\text{reg}}$, then it should either be holdsAt or broken.

Where such a rule has ‘permitted’ in the head, it is a positive authorization rule; otherwise, it is known as a negative authorization rule. (Additional constraints of local stratification are often used to demonstrate the expressiveness of security languages. Our formalism can represent all policies of this type we have examined. Chinese Wall scenarios can be modelled easily, by considering the system history.

**Example 1** “A mobile node may delete classified data if it sends a notification to the supplier of the data 10 minutes in advance, and the supplier does not respond to the notification asking the node to retain the data.”

We represent this as follows:

```plaintext
permitted(N, D, delete, T) ←
holdsAt(fileDesc(D, class), Tn),
holdsAt(owner(D, O), Tn),
do(N, O, notify(delete, D), Tn),
T = Tn + 10,
not reqInBetween(O, N, retain(D), Tn, T).
```

The predicate `reqInBetween` is related to the operator `SINCE` of temporal logics [15]; we have found such a predicate useful on several occasions. To capture its semantics, the following rule is always included in our framework:

```plaintext
reqInBetween(Sub, Tar, Act, T') ←
req(Sub, Tar, Act, T), T' ≤ T, ≤ T.
```

An instance `reqInBetween(Sub, Tar, Act, 0, T)` means that a request (with the relevant parameters) was made at some time before $T$; this is related to the modal temporal operator expressing that a property held at some previous time.

Separation of duty (SoD) [3] and Chinese Wall policies [8] are often used to demonstrate the expressiveness of security policy languages. Our formalism can represent all policies of this type we have examined. Chinese Wall scenarios can be modelled easily, by considering the system history.

**Example 2** “A person cannot assist in a medical situation once he has taken part in surveying a contaminated area.”

This can be represented as the following negative authorization rule:

```plaintext
denied(Sub, M1, assist, T) ←
do(Sub, M2, assist, T'),
holdsAt(activity_type(M1, medical), T),
holdsAt(activity_type(M2, survey(A)), T'),
holdsAt(area_classify(A, contaminated), T'),
T' < T.
```

Simple dynamic SoD policies that define mutually exclusive role activation are handled by restricting role activations as follows:

```plaintext
denied(Subject, roles, activate(role_a), T) ←
holdsAt(isActivated(Subject, role_b), T).
```

```plaintext
denied(Subject, roles, activate(role_b), T) ←
holdsAt(isActivated(Subject, role_a), T).
```

Other classes of SoD policy can be captured in a similar manner.

When gathering together authorization rules to form an authorization policy, it is normal to include a number of more general rules. These can be used to state whether a request to perform an action is accepted (and the action performed) by default if there is no explicit permission in the policy rules; or whether explicit permission is required; what response (if any) should be given if an action is denied, and so on. We see it as a virtue of our framework that many different rules which embody the action of the PEP can be represented, and that no one approach is fixed as part of the formalism. This flexibility is crucial if we need to cover the behaviour of different policy systems in heterogeneous environments. Consider the three example availability rules in Table 2. The basic availability rule is more stringent: an
action is permitted by the PEP only when it has been positively permitted by the PDP—similar to [23]. The positive availability rule is less strict: actions are executed so long as they have not been expressly denied by the policy rules. The negative availability rule states that an output deny predicate is true whenever an action is explicitly denied by the policy rules. The effects of this deny predicate can then be modelled—a typical use may be to cause logs of denials of requests to be kept in the system.

**Definition 3** A policy regulation rule has one of the predicates do or deny in the head and a body given as in Definition 2.

Many more policy regulation rules are possible than those given as examples in Table 2; all are optional inclusions in an authorization policy.

**Definition 4** An authorization policy is a set II of authorization rules, with the clause defining reqInBetween, and a set of policy regulation rules, such that II is locally stratified.3

Notice that it is possible to add general authorization rules to a policy, enabling a representation of very fine-grained defaults controlling responses to requests. For example, if a user belongs to the root system group, one may want to permit all the actions of that user by default, unless they are explicitly denied:

\[
\text{permitted(Sub, Tar, Act, T)} \leftarrow \text{group(Sub, root), not denied(Sub, Tar, Act, T).}
\]

### 3.3 Obligations

The obligations we represent are on a subject to perform an action on a target, a class which includes a large number of practical obligation policies [17]. As in most (if not all) deontic logics, obligations may be fulfilled or not, allowing us to represent the behaviour of systems of which humans are a part. We present a simplified version of our treatment of obligations here, in which the period during which an action should be performed is delimited by explicit reference to time. Our general language also allows the user to specify events or actions as these delimiters.

**Definition 5** An obligation policy rule is a formula

\[
\text{obl(Sub, Tar, Act, T, T1, T2)} \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.
\]

where the conditions 1–4 as for Definition 2 hold, with the addition that \(T_e\) and \(T_s\) should be variables of type Time.

3A set of rules is locally stratified if in the set of all ground instances of the rules (i.e., where all variables are replaced by all their possible values) there is no head of a rule that depends directly or indirectly on the negation of itself. Testing for local stratification is, in general, computationally hard; but large classes of rules can be identified as locally stratified easily based on the time index [50].

3Positive obligation policy rules will always include constraints which make this true.)

Two domain-independent rules accompany the obligation rules, defining the fulfillment and the violation of an obligation:

\[
\begin{align*}
&\text{fulfilled(Sub, Tar, Act, T_e, T_s, T)} \leftarrow \\
&\text{obl(Sub, Tar, Act, T_e, T_s, T1)}, \quad \text{do(Sub, Tar, Act, T')} \\
&\text{not cease_obl(Sub, Tar, Act, T1, T, T')}, \quad T_s \leq T_e < T, \quad T' < T.
\end{align*}
\]

\[
\begin{align*}
&\text{violated(Sub, Tar, Act, T_e, T_s, T)} \leftarrow \\
&\text{obl(Sub, Tar, Act, T, T1)}, \quad \text{not cease_obl(Sub, Tar, Act, T1, T, T)}, \quad \text{do(Sub, Tar, Act, T')} \\
&\quad T_s \leq T_e < T, \quad T' < T.
\end{align*}
\]

An obligation is fulfilled when the action a subject has been obliged to perform is executed (notice that the do in the body of the rule here means that the execution of such an action must first be enforced by the PEP, based on the decisions of the PDP). An obligation is violated when no such action occurs. The rules for fulfilled and violated use cease_obl as a subsidiary predicate, defined by the following rules:

\[
\begin{align*}
&\text{cease_obl(Sub, Tar, Act, T1, T, T)} \leftarrow \\
&\text{do(Sub, Tar, Act, T')} \quad T_s < T_e < T \\
&\text{cease_obl(Sub, Tar, Act, T1, T, T, T')} \leftarrow \\
&\text{do(Sub}, \text{Sub, revoke(Sub, Tar, Act, T, T)} \quad T_s < T_e < T.
\end{align*}
\]

cease_obl is a predicate used to mark the fact that an obligation has either been fulfilled or revoked. There are therefore two clauses defining cease_obl. The cease_obl rule for revocation makes use of the revoke members of the sort Action, mentioned in Section 3.1; revocation occurs when the PDP has authorized the request for a revocation action. The subject requesting a revocation might be the one bound by the obligation, a central administrator in the system, or an entirely different agent. The parameters of the revoke argument identify the obligation to be revoked.

**Example 3** “A connecting node is obliged to re-identify itself within five minutes of establishing a connection to the server; otherwise the server must drop the connection within one second.”

This example in fact includes two obligations: one on the node making the connection, and one on the server, which...
must drop the connection if the node does not fulfil its obligation. They can be formalized as follows:

\[
\begin{align*}
\text{obl}(U, \text{serv}, \text{sub2ID}(U, \text{serv}), T + 300, T + 300) & \leftarrow \\
\text{holdsAt}(\text{node}(U), T), \text{do}(U, \text{serv}, \text{connect}(U, \text{serv}), T).
\end{align*}
\]

\[
\begin{align*}
\text{obl}(\text{serv}, \text{connect}(U, \text{serv}), T, T_n + 1, T_n) & \leftarrow \\
\text{violated}(U, \text{serv}, \text{sub2ID}(U, \text{serv}), T, T_n, T_n).
\end{align*}
\]

The EC predicate \text{holdsAt} is used to represent dynamic properties of the system: in this case, which nodes are registered. The obligation begins just \(\epsilon\) seconds after the server connects to the node.

**Definition 6** An obligation policy \(\Pi\) is a set of obligation rules, with the ‘fulfilment’, ‘violation’ and ‘cease obl’ rules, such that \(\Pi\) is locally stratified.

**Definition 7** A security policy \(\Pi = \Pi_s \cup \Pi_o\) is any union of an authorization policy \(\Pi_s\) and an obligation policy \(\Pi_o\).

### 3.4 Domain Models

We use the Event Calculus (EC) to represent and reason about changing properties of the domains regulated by policies. The EC is a well-studied, logic-based formalism, variants of which exist both as logic programs and in first-order logical axioms (using a second-order axiom to enforce a circumscription semantics). It has the ability to represent concisely the effect of actions on properties of a system, and built-in support for the default persistence of fluents. The EC is used to model and implement many dynamic systems (see [1] for a recent example, or [25] for general references).

In the EC, effects of events or occurrences are defined by two predicates \text{initiates} and \text{terminates}. \text{initiates} describes which state properties are caused to hold by an event; and \text{terminates} describes which properties cease holding after an event. The rules which define the two predicates can have conditions. Users may also define a number of state constraints, which have atoms of the predicate \text{holdsAt} in the head, and which represent that a state has a given property, if the same state has certain other properties. Core axioms are then added, common to any EC formalization, to relate the behaviour specifications of the \text{initiates} and \text{terminates} axioms to state properties. These core axioms, the set \(\text{EC}\), are shown below.

\[
\begin{align*}
\text{holdsAt}(F, T) & \leftarrow \\
\text{initially}(F), \not\text{broken}(F, 0, T).
\end{align*}
\]

\[
\begin{align*}
\text{holdsAt}(F, T) & \leftarrow \\
\text{initiates}(\text{Sub:Tar:Act}, F, T), \\
T_n < T, \\
\text{do}(\text{Sub, Tar, Act}, T_n), \\
\not\text{broken}(F, T_n, T).
\end{align*}
\]

holdsAt\((F, T) \leftarrow \)

\[
\text{initiates}(\text{Event, } F, T_s), \\
T_s < T,
\]

\[
\text{happens}(\text{Event}, T_s), \not\text{broken}(F, T_s, T).
\]

broken\((F, T_s, T) \leftarrow \)

\[
\text{terminates}(\text{Sub:Tar:Act, } F, T'), \\
\text{do}(\text{Sub, Tar, Act}, T'), \\
T_s < T' < T.
\]

\[
\begin{align*}
\text{broken}(F, T_s, T) & \leftarrow \\
\text{terminates}(\text{Event, } F, T'), \\
\text{happens}(\text{Event, } T'), \\
T_s < T' < T.
\end{align*}
\]

The first clause (3) specifies that a changeable property of the system holds at time \(T_s\); if that property held at time 0 and nothing disturbed its default persistence. The next two clauses (4 and 5) define how a fluent representing a changeable property comes to be true: by being initiated, either as a consequence of an action enforced by the PDP/PEP, or by being the result of an unregulated event occurring in the system. The final two clauses (6 and 7) represent how an event disturbs the persistence of a fluent, preventing its truth from persisting over time; again, there is a clause for disturbance caused by enforced regulated actions, and another for disturbance caused by unregulated events. For more details see the original formulation in [22], or for recent approaches, [25].

In order to improve the analysis algorithm, we separate the predicates used to represent the static portion of the system from the predicates concerning the changing properties. The former are contained in \(\mathcal{L}_{\text{stat}}^D\). As these static properties either hold for all times or none, there is no need to model the effects of actions on their holding, and thus no need to use the EC to reason about them.

**Definition 8** A domain description \(D = \text{EC} \cup D'\) contains the core axioms \(\text{EC}\) and a set \(D'\) of formulas of any of the three forms: a static domain axiom

\[
A \leftarrow L_1, \ldots, L_n.
\]

such that \(A\) is an atom and \(L_1, \ldots, L_n\) are literals of predicates in \(\mathcal{L}_{\text{stat}}^D\); a state constraint

\[
\text{holdsAt}(F, T) \leftarrow L_1, \ldots, L_n,
\]

in which the \(L_1, \ldots, L_n\) are literals of predicates in \(\mathcal{L}_{\text{stat}}^D \cup \{\text{holdsAt}\}\) and all Time variables in the \(L_i\) are equal to \(T\); or an initiates or terminates axiom

\[
\text{initiates}(X, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.
\]

\[
\text{terminates}(X, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.
\]

such that:

- \(\text{initiates}(X, F, T), \text{terminates}(X, F, T) \in \mathcal{L}_{\text{EC}}\).
- Each \(L_i\) is a literal of an atom in \(\mathcal{L}_{\text{stat}}^D\), or else a literal of the predicate \text{holdsAt}; each \(C_i\) is a time constraint.
Each variable appearing in a time constraint must also appear somewhere other than in a time constraint.

For any time argument $T_i$ of an $L_i$, we must have $C_1 \land \cdots \land C_n \models T_i \leq T$.

Domain descriptions must be locally stratified.

As an example of a common system description in policy representation, we consider a simple subset of the RBAC model [13]. We represent user-to-role assignment by the fluent $\text{hasRole}(\text{Subject}, \text{Role})$ and permission-to-role assignment by $\text{hasPerm}(\text{Role}, \text{Resource}, \text{Action})$. The access control can then be expressed by the following axiom:

permitted(\text{Sub}, \text{Resource}, \text{Act}, T) \leftarrow
\text{holdsAt}(\text{hasRole}(\text{Sub}, \text{Role}), T),
\text{holdsAt}(\text{hasPerm}(\text{Role}, \text{Resource}, \text{Act}), T).

The following axioms capture the role hierarchy inheritance:

\text{holdsAt}(\text{subrole}(R, R'), T) \leftarrow
\text{holdsAt}(\text{contains}(R', R), T).

\text{holdsAt}(\text{subrole}(R, R'), T) \leftarrow
\text{holdsAt}(\text{contains}(R', R), T),
\text{holdsAt}(\text{subrole}(R', R), T).

\text{holdsAt}(\text{hasRole}(U, R), T) \leftarrow
\text{holdsAt}(\text{hasUser}(R, U), T).

\text{holdsAt}(\text{hasRole}(U, R), T) \leftarrow
\text{holdsAt}(\text{hasUser}(R, U), T),
\text{holdsAt}(\text{subrole}(R', R), T).

One advantage of the EC is that using the same formalism we can also express the administration of RBAC (ARBAC—see e.g. [29]). First, we use EC rules to model the effects of user-role assignments (i.e. adding or removing assignments):

\text{initiates}(S:R:assignUser(U), \text{hasUser}(R, U), T).
\text{terminates}(S:R:unassignUser(U), \text{hasUser}(R, U), T).

We also need to model the effects of the role-permission assignments:

\text{initiates}(S:R:assignPerm(T, A), \text{hasPerm}(R, T, A), T).
\text{terminates}(S:R:unassignPerm(T, A), \text{hasPerm}(R, T, A), T).

And finally, operations to the role hierarchy:

\text{initiates}(\text{Admin}:R:assignRole(R'), \text{contains}(R, R'), T).
\text{terminates}(\text{Admin}:R:removeRole(R'), \text{contains}(R, R'), T).

When modeling an instance of an ARBAC system, we need to define roles, user-to-role assignment and permission-to-role assignment for administrators to perform operations on the RBAC system. This can be done by creating the correct role hierarchy, user-role assignments and role permissions. For example,

initially(\text{hasUser}(\text{admin}, \text{alice})).
initially(\text{hasPerm}(\text{admin}, R, \text{assignUser}(U))).
initially(\text{hasPerm}(\text{admin}, R, \text{assignPerm}(T, A))).
initially(\text{hasPerm}(\text{admin}, R, \text{removeRole}(R'))).

We now show a system trace. In general, these are sequences of actions performed in the system, which determine the changes in state properties. The actions can be either policy-governed (as in the present case), or outside the control of the policy system (for example, a human pressing a button). The following is an example of a system trace in which we assume there are two roles (Medical Aid and Field Surgeon) present in the system, but not yet in any hierarchical relationship with each other.

\text{do}(\text{alice:medical_aid:assignRole(field_surgeon, 0)}).
\text{do}(\text{alice:medical_aid:assignPerm(S, initialExamine, 1)}).
\text{do}(\text{alice:field_surgeon:assignUser(daneeka, 2)}).
\text{do}(\text{alice:field_surgeon:assignPerm(S, operate, 3)}).
\text{do}(\text{alice:medical_aid:assignUser(duckett, 4)}).

In this trace, the administrator Alice adds the role Field Surgeon to be a sub-role to that of Medical Aid. The latter role is then assigned the permission to perform the initial examination of any soldier. Doctor Daneeka is assigned to be a user in the role of field surgeon, and the role of field surgeon is assigned permission to operate. Finally, Nurse Duckett is assigned to the role of medical aid. Here, as the field surgeon role is a sub-role of that of medical aid, the permission of performing the initial examination is inherited by the role of field surgeon (according to the logic of the axioms expressing inheritance given previously) and thus conferred on Doctor Daneeka.

Below is a simple example of static Separation of Duty (SOD) in this RBAC model, between the roles of members of the Medical Aid team role and the Security Officer team role.

\text{denied}(\text{Admin, sec_officer, assignUser}(U, T) \leftarrow
\text{holdsAt}(\text{hasUser}(\text{medical_aid}, U), T).

\text{denied}(\text{Admin, medical_aid, assignUser}(U, T) \leftarrow
\text{holdsAt}(\text{hasUser}(\text{sec_officer}, U), T).

Axioms for constraints and sessions and other administrative operations in RBAC and ARBAC can also be expressed in our domain description model.

### 3.5 Domain-constrained Policies

We bring all the previous definitions together, to describe our complete models of systems constrained by policies.

**Definition 9** A domain-constrained policy $P = \Pi \cup D$ is the union of a security policy $\Pi$ and a domain description $D$.

(Not that if $\Pi$ and $D$ are locally stratified, then so is $\Pi \cup D$.)
We use the standard stable model semantics [14] of logic programs. To capture the operational model we start with any set $\Delta^D$ of ground instances of non-regulatory predicates from the set \{\text{initially, happens}\} \cup L_{\text{stat}}^D and any set $\Delta^*$ of ground instances of the regulatory predicate req. The sets $\Delta^D$ and $\Delta^*$ represent information about the inputs to the system, about events which are not controlled by the PDP/PEP, and information about the system’s initial state, together with facts about the unchanging (static) properties of the regulated system. In general, different sets $\Delta^D, \Delta^*$ can be thought of as representing different initial configurations and runs through the system which is governed by our policy mechanism.

**Definition 10** Let $P$ be a domain-constrained policy (see Definition 9). Then, a policy-regulated trace is the stable model of ground$(P \cup \Delta^D \cup \Delta^*)$. We let model$(P \cup \Delta^D \cup \Delta^*)$ refer to the (unique) stable model of ground$(P \cup \Delta^D \cup \Delta^*)$.

### 4. POLICY ANALYSIS

In this section we illustrate a number of different analysis tasks which can be performed within our framework.

(i) **Modality Conflicts.** The task of analysing a domain-constrained policy $P = \Pi \cup D$ to see, for instance, whether there are no modality conflicts (e.g., permits and denials over the same resource, or obligations over resources for which a subject has no authorizations), is an instance of the more general task of determining whether (stable) models of the domain-constrained policy verify a number of properties. For instance, we may wish to prove the following freedom for a particular kind of modality conflict:

\[
\forall T( - (\text{permitted}(\text{sub, tar, act, } T) \\
\text{\& denied}(\text{sub, tar, act, } T)))
\]  

(8)

This formula states that for all times, it is not true that an action is both permitted and denied. If we cannot prove this, then we wish to have diagnostic information about the circumstances in which it fails to be true. Checking whether the system verifies this property converts into the task of checking whether there are inputs $\Delta^D$ and $\Delta^*$ (as described in Section 3.5) such that the property is not true, i.e., whether

\[
\text{model}(P \cup \Delta^D \cup \Delta^*) \models \exists T(\text{permitted}(\text{sub, tar, act, } T) \\
\text{\& denied}(\text{sub, tar, act, } T))
\]

This is equivalent to showing that the previous formula (8) is false, and can be solved using Abductive Logic Programming (with constraints—ACLP), which computes the sets $\Delta^D$ and $\Delta^*$. The output of the algorithm will be these sets together with a number of constraints (expressed as equalities and inequalities) on the possible values of the time-arguments appearing in the answers. In our implementation we currently use an abductive constraint logic programming proof procedure based on that found in [21]. (For the details of the algorithm, see Appendix A.)

(ii) **Illustration of trace abduction.** One of the example scenarios we have been using to test our framework concerns a natural-disaster rescue domain, in which a team of medics must react to injuries incurred by people caught in an earthquake. We will show a sample analysis, together with the diagnostic information which our prototype system provides. Here is a small subset of the policies, in natural language and then in our policy representation language:

- **Nobody may move a patient with spinal injuries**
  \[
  \text{denied}(\text{Sub, Tar, move}(L), T) \leftarrow \\
  \text{holdsAt}((\text{is}_\text{injured}(\text{Tar, spinal}), T).
  \]
  (9)

- **Medics are allowed to move a patient with a spinal injury if they are on a spine board.**
  \[
  \text{permitted}(\text{M, Tar, move}(L), T) \leftarrow \\
  \text{holdsAt}((\text{is}_\text{injured}(\text{Tar, spinal}), T), \\
  \text{holdsAt}(\text{on_spine_board}(\text{Tar}), T).
  \]
  (10)

- **Injuries who are located in a house at risk of collapse must be moved to hospital within 10 mins by a medic**
  \[
  \text{obl}(\text{M, Tar, move(hosp)}, T_e, T_s, T_e) \leftarrow \\
  T_e = T_s + 10. \text{\& holdsAt}(\text{at}\_\text{risk}(\text{Tar, } H), T_s), \\
  \text{holdsAt}(\text{at}\_\text{risk}(H), T_s), \\
  \text{happens}(\text{find}(X, T_s), T_s), \\
  \text{holdsAt}((\text{is}\_\text{injured}(\text{Tar, InjuryType}), T_s).
  \]
  (11)

In addition to policies such as these, the example domain also includes formulas which describe the effects of actions, expressed in the EC formalism described in Section 3.4. They specify under what circumstances a house is at risk of collapsing, how this can cause injuries to individuals, the remedial actions medics can take to treat injuries, and so on. We do not present these details here.

Even with the few simple policies presented above, a number of interesting analyses are possible. Are there situations in which a medic has an obligation which it would be impossible to fulfill, because of the presence of a conflicting authorization policy? There are several different interpretations of this query, one of which is the following:

\[
\text{obl}(\text{Sub, Tar, Act, } T_s, T_e, T_{\text{med}}) \leftarrow \\
\text{not cease_obl}(\text{Sub, Tar, Act, Tinst, } T_s, T_e, T), \\
\text{\&\& denied}(\text{Sub, Tar, Act, } T) \leftarrow T_s < T.
\]

(12)

If there are values of the unbound variables which makes the above conjunction true, this means there is a time at which an obligation is binding on a Sub, but at which there is a negative authorization policy, stating that the Sub will be denied access to perform the action. This query can be solved in our framework by the abductive algorithm; one of the answers returned shows the following groundings for the variables in (12):

\[
\text{obl}(\text{medic, alice, move(hosp)}, 1, 1, 11) \leftarrow \\
\text{\&\& not cease_obl}(\text{medic, alice, move(hosp)}, 1, 1, 11, 2), \\
\text{\&\& denied}(\text{medic, alice, move(hosp)}, 2) \leftarrow 1 < 2.
\]

(13)
The abducted atoms, the union of the sets $\Delta^D$ and $\Delta^r$, were:

\[
\{ \text{initially(at\_risk(house3))}, \\
\text{initially(at(alice, street))}, \\
\text{happens(walk(alice, house3), 0)}, \\
\text{happens(injure(alice, spinal), 1)} \}
\] (14)

These atoms represent a series of events and actions in the system, together with system’s initial configuration, which will lead to the presence of the modality conflict represented by (13). They show that if alice is initially in the street, then walks to house3—which is at risk of collapse—and subsequently has a spinal injury, there will be an obligation on the medics to move him back to the hospital, but a denial of permission to make that movement.

Policy authors can look at this output and take necessary actions. One response might be to introduce exceptions to the policy rules (9) in order to avoid the modality conflict, but for the purpose of illustrating another capability of our analysis framework, we consider a different possibility.

(iii) Constrained search and multiple solutions. Suppose the policy author, familiar with some details of the system which is being controlled by the policy, notes that the sample trace above, which gives rise to a modality conflict of the kind queried, has the event

\[
\text{happens(injure(alice, spinal), 1)}
\]

It may be that this event would never occur in the real system—let us say e.g. that it is known that alice had been outfitted with a specially-reinforced protective suit for the exploration of dangerous buildings. The system model, as an abstraction of the real world, might not contain this information, but the policy author is aware of it. In this circumstance, our analysis system allows its user to modify the original query (12), to introduce a constraint stating that alice is not injured. The modified query would be:

\[
\begin{align*}
\text{obl}(Sub, Tar, Act, T_s, T_e, T_{\text{sub}}) \\
\land \text{not cease\_obl}(Sub, Tar, Act, T_{\text{sub}}, T_s, T_e) \\
\land \text{denied}(Sub, Tar, Act, T) \land T_s < T \\
\land \text{not holdsAt}(\text{injure}(alice, spinal), T)
\end{align*}
\] (15)

This includes the relevant constraint, as the final literal. Its inclusion prevents the first solution, the abducted atoms (14), from being found by our analysis procedure; alternative solutions are explored, such as that represented by the following sample sets of abducted atoms. First:

\[
\{ \text{initially(at\_risk(house3))}, \\
\text{initially(at(bob, street))}, \\
\text{happens(walk(bob, house3), 0)}, \\
\text{happens(injure(bob, spinal), 1)} \}
\] (16)

This solution has ignored alice, and suggested that bob may find himself in the same situation as that represented in (14). Another solution:

\[
\{ \text{initially(at\_risk(house3))}, \\
\text{initially(at(bob, street))}, \\
\text{initially(at(alice, street))}, \\
\text{initially(injured(bob, leg\_break))}, \\
\text{happens(carry(alice, bob, house3), 0)}, \\
\text{happens(injure(bob, spinal), 1)} \}
\] (17)

This is a different analytic trace: both alice and bob are in the street to start with, and bob has a leg injury. At time 0, alice then carries bob to the house which is at risk, whereupon bob is spinally injured, and our modality conflict arises.

(iv) Separation of Duty. Separation of duty has been mentioned several times in previous sections. Checking for properties related to it follows the same pattern as for other properties. For example, violations of dynamic SoD can be checked with:

\[
\begin{align*}
\text{model}(P \cup \Delta^D \cup \Delta^r) & = \\
\exists T(\text{permitted}(\text{sub}, \text{roles}, \text{activate}(\text{role}_A), T) \\
\land \text{permitted}(\text{sub}, \text{roles}, \text{activate}(\text{role}_B), T))
\end{align*}
\]

This query states that there is a time at which sub is permitted both to activate role A, and role B.

(v) Coverage gaps. We can also—as mentioned in the introduction—perform coverage analysis, in which a policy is checked, against the background of a particular system, to see whether there are system traces which give rise to states in which a request for action has no relevant policy rule covering it. Coverage gap analysis has two types: the first involves checking the explicit policy rules for gaps, without taking into account the default logic of the policy regulation rules; the second adds the policy regulation rules. We make remarks on each kind below.

The first kind of coverage gap analysis considers situations in which a request for the performance of an action is received, but there is no explicit permission or denial implied by the authorization policy rules of the system. This form of analysis can be performed using a query of this form:

\[
\begin{align*}
\text{model}(P \cup \Delta^D \cup \Delta^r) & = \\
\exists \text{Sub, Tar, Act, } T (\text{req(Sub, Tar, Act, } T) \\
\land \neg \text{permitted(Sub, Tar, Act, } T) \\
\land \neg \text{denied(Sub, Tar, Act, } T))
\end{align*}
\]

As with the modality conflict analysis above, diagnostic trace information is supplied.

Any action to be performed on a person who is explicitly stated not to have a spinal injury, for example, will not be covered by the set \{(9), (10), (11)\}; a query of, say:

\[
\begin{align*}
\text{req(Sub, Tar, triage, } T) \\
\land \neg \text{permitted(Sub, Tar, triage, } T) \\
\land \neg \text{denied(Sub, Tar, triage, } T) \\
\land \text{not holdsAt(is\_injured(Tar, spinal), } T)
\end{align*}
\] (18)

finds many answers, depending on the number of subjects and targets in the domain.
This kind of coverage gap analysis concerns the absence of what we might call explicit authorization decisions for a request for action: the case where there is no positive or negative authorization policy rule covering the case in question. Whether or not an action is enforced by the PEP following a request, however, is decided in our framework by the conjunction of these authorization policy rules and the rules governing default availability, which have do in their head (see Section 3.2). Thus, a situation may arise in which, whilst there is no decision on an access request explicitly forced by the authorization rules, the request is still allowed or denied, because of the presence of a policy regulation rule such as:

\[
do(Sub, Tar, Act, T) \leftarrow \\
\text{not} \ \denied(Sub, Tar, Act, T).
\]

This leads us, therefore, to the second general type of coverage gap analysis: that which asks for requests which would be allowed, but not as a result of the explicit authorization policies, but merely as a consequence of the default permissions of the system. These cases can be found by a query such as:

\[
\text{req}(Sub, Tar, Act, T) \\
\land \text{not} \ \permits(Sub, Tar, Act, T) \\
\land \do(Sub, Tar, Act, T).
\]

Again, this general form of query can be made specific to individual users or actions, or types of users and actions.

(vi) Behavioural simulation. Note that to some degree the example query (18) mixes coverage gap analysis with behavioural simulation. In the latter, a typical query would involve inputting a series of events and requests which might occur in a system and analysing, deductively, what permissions were granted, and what the resulting system state is. In query (18), we specify properties of the system trace by including

\[
\text{not} \ holdsA(is\_injured(Tar, spinal), T)
\]

excluding some system traces but allowing others.

A more straightforward example of behavioural simulation is shown by considering the sample trace of do atoms given towards the end of Section 3.4, in which an administrator Alice assigns a number of users and permissions to roles. Given that simulation of the behaviour of the system, a user can query whether, for instance, Nurse Duckett is permitted to perform an initial examination of a patient at time 2:

\[
\permits(duckett, P, \text{initialExamine}, 2).
\]

This would be answered negatively; although the permission for initial examinations has been assigned to the role of medical aid by time 2, Nurse Duckett has not yet been given the role. If the same query is posed after the system has further evolved—say, at time 5—then the authorization would be granted.

(vii) Policy comparison. Finally, we make some brief remarks about policy comparison. In this form of analysis, we can check to see whether one policy is included in another, or whether one implies another, whether they are equivalent, and so on.

The fact that our analysis framework allows us to test for these inclusions means that an engineer seeking to modify a policy for a given system can prove whether or not his modifications would have any effect, or whether the addition, say, of a proposed new authorization rule is in fact redundant. Suppose, for example, that the current policy set is given by \(\{(9), (10), (11)\}\), as in our running example, and let us say that the proposed new positive authorization rule is

\[
[\text{Patients in category ‘z’ are allowed to be moved}]
\]

\[
\permits(M, Tar, move(L), T) \leftarrow \\
\text{holdsA(category(Tar, z), T)}.
\]

Suppose that the domain is such that a person is classified as in the z category if and only if they have a spinal injury. For example, the domain description may contain the following rules:

\[
\text{initiates}(\text{injure}(Tar, \text{spinal}), \text{is\_injured}(Tar, \text{spinal}), T).
\]

\[
\text{terminates}(\text{cure}(Tar, \text{spinal}), \text{is\_injured}(Tar, \text{spinal}), \text{category}(Tar, z), T).
\]

In this case, adding the rule (19) to the policy would have no effect. If \(\Pi_2\) denotes the policy set \(\{(9), (10), (11)\}\) and \(\Pi_1\) denotes the set \(\{(9), (10), (11), (19)\}\), with \(D\) being the full version of our system description, including the initiates and terminates rules above, then we would receive a positive answer to the query \(\Pi_1 \subseteq D \Pi_2\), indicating that relative to the particular system description, the rule (19) is redundant. (The relationship \(\Pi_1 \subseteq \Pi_2\) is defined to mean that, given a domain description \(D\), for all system traces, any permissions, denials, or obligations which are implied by the policy \(\Pi_1\) are also implied by the policy \(\Pi_2\).)

One way of checking whether \(\Pi_1 \subseteq \Pi_2\), using our analysis framework, is to relativize the policy representation languages, so that the state regulatory predicates receive a subscript. Policies from \(\Pi_1\) would then be written using \(\permits_1\), \(\denied_1\), and so on; and policies from \(\Pi_2\) would be written using \(\permits_2\), \(\denied_2\), etc. Clauses which are common to all domain descriptions or security policies, such as the EC axioms or the rules giving the meaning of fulfilled and violated ((1) and (2)), would be replaced by two versions: one containing the subscript 1 on the state regulatory predicates, and one containing the subscript 2. If the three queries:

\[
\permits_1(Sub, Tar, Act, T) \\
\land \text{not} \ \permits_2(Sub, Tar, Act, T),
\]

\[
\denied_1(Sub, Tar, Act, T) \\
\land \text{not} \ \denied_2(Sub, Tar, Act, T),
\]

\[
\text{obl}_1(Sub, Tar, Act, T, T) \\
\land \text{not} \ \text{obl}_2(Sub, Tar, Act, T, T, T)
\]

each returned no answers, given a domain description \(D\), this would be a proof that \(\Pi_1 \subseteq \Pi_2\).

Further, as with previous forms of policy analysis, the queries can be made as general or specific as the analysis task demands. It is possible to ask, for instance, whether a policy \(\Pi_1\) extends the obligations of users on the medical
team to move patients, compared to policy $\Pi_2$, by a query such as:

\[
\text{obh}((\text{Sub}, \text{Tar}, \text{move}(L)), T, T, T) \\
\land \neg \text{obl}_2((\text{Sub}, \text{Tar}, \text{move}(L)), T, T, T) \\
\land \neg \text{holdsAt}(\text{team}(\text{Sub}, \text{medical}), T)
\]

Further examples and system traces can be found on the website for the implementation.

4.1 Termination and Complexity

We consider termination and computational complexity properties for two aspects of our formal framework—the runtime evaluation of policy rules, and the offline analysis of policies accomplished using the abductive approach just described. By evaluation, we mean the determination of answers to queries about which actions are permitted, denied, or constrained by obligations, using SLNDF.

The language we use (the sorts Subject, Target, Action, Fluent, Event) is finite. If we further stipulate that the models of a domain-constrained policy $P = \Pi \cup D$ must be such that in the security policy component $\Pi$, there is a maximum value $t$ such that whenever a body of a policy rule is satisfied true by the model, all time indices must belong to some interval $[t, t + \ell]$, and if only a finite number of actions can occur within any given finite time, then a finite amount of information needs to be stored about the system evolution in order to evaluate policies. For example, if there is a rule

\[
\text{permitted}((\text{Sub}, \text{Tar}, \text{Act}, T) \leftarrow \\
\text{holdsAt}((T'), T) \\
T = T' + 10,
\]

in the policy, we know we must record information about whether the fluent $f$ holds 10 seconds in the past; beyond 10 seconds, we may not care (depending on the other policies in $\Pi$) what happens to $f$. For any given domain-constrained policy, a bound on the amount of domain-dependent information which needs to be stored can be calculated, based on the language, the policy set, and the domain description.

In order to ensure that the evaluation of policy rules expressed in our formalism terminates, and that this procedure runs efficiently, we must ensure that there are no circular dependences amongst the members of our security policy component, and thus there is a guarantee of soundness. In the most general case, our language is expressive enough to allow the presence of circular dependences amongst literals in policy rules, and thus there is not, at the most general and unrestricted level, a guarantee of termination and therefore of completeness. However, if we make a further restriction that, in addition to a maximum time interval $[t, t + \ell]$ in the body of policy rules (which we made for the case of policy evaluation, above), there is also a maximum time in the past that we will recourse over in our analyses, we can ensure termination and completeness. Further, our language is expressive enough to represent, and our analysis algorithms powerful enough to solve, classes of problem such as the ones identified in [29] and in [17] that are NP-hard, giving an indication of the computational complexity of the abductive analysis we use. Having abduction as a uniform mechanism for solving analysis problems will let us work on optimizations and approximations for abductive procedures semi-independently of the analysis. The current implementation of abduction we use is more general than that strictly required by our analyses.

4.2 Implementation

A prototype implementation of our formal analysis framework is freely available to download.\footnote{Reference removed through anonymity requirement on submission.} The implementation uses the open-source abductive constraint logic programming ASYSTEM [31]. Tests have enabled us to find modality conflicts, coverage problems, and other interesting properties of policies in conjunction with system descriptions, such as those earlier in this section.

The ASYSTEM is based on finite domains. For this reason, we adapted our axioms to work on an integer base for Time, and chose a maximum time to consider in order to make the Time domain finite. In all cases we have examined, analysis results under these modifications would hold under the original version of the axioms with $\mathbb{R}$. The abductive logic-programming framework we use is modular, so that a solver based on the real numbers could simply be ‘plugged in’ to the algorithm instead. This is an area of current investigation.

5. CONCLUSION

A formal policy framework must incorporate obligations as well as authorizations, include an analysis component using information about changing system state for accurate proof of significant properties, provide rich diagnostic information as output, separate the representation of system from policy, and include policies which depend on each other and contain fine-grained defaults. Many languages aim to achieve some of these goals, but none succeed in achieving all in a way which balances expressiveness with efficiency of evaluation and analysis.

Our framework was designed to meet these requirements.
We defined the structure of the policy language, and described how we use the EC to depict and reason about changing properties of the system. We gave examples of authorization and obligation rules, and described how abductive algorithms lying at the heart of our framework can be used in the analysis, discussing the current implementation.

By separating the representation of the laws of system evolution, and constraints on the system state, from the authorizations and obligations which define policy decisions we gain clarity in the representation but also the ability to switch domain descriptions easily and study the behaviour of policies on different systems.

The choices we have made in the design of the language show that it is possible to encode subtle default relationships and decisions without sacrificing efficiency, readability or concision. The use of temporal constraints and an explicit representation of time has enabled us to express complex dependences of policy decisions on changing system states, as well as on other policies.

Abductive Constraint Logic Programming is a suitable paradigm for the kinds of analysis task we wish to perform on policies. We have used it successfully to provide rich diagnostic information on the system traces and initial conditions which give rise to properties of policies in heterogeneous environments: in this way, the use of ACLP with the Event Calculus and separable policies and system representations has been shown to be an effective combination for policy analysis. We have also used abduction, in our analysis framework, to fill in a partially-specified system, so that initial conditions which might give rise to e.g. modality conflicts are generated as hypotheses.

Further work is ongoing both at the implementation and at the theoretical level. At the moment, all suitable ACLP systems use integers as a basis of their constraints, but the modularity of the abductive approach we have taken means that an implementation based on reals is entirely feasible. We are also completing the work on translations between our framework and other languages for policies representation. We currently have translation schemes for Ponder2 [28], and are working on schemes for XACML [27] and others.

Our broader objective is to define a refinement framework, of which the analysis framework in the current paper will form a part. Within this context, an expressive abstract policy language is necessary both to represent a broad spectrum of high-level policies but also to accommodate different concrete mechanisms on which policies need to be implemented. Our previous work on policy refinement [ref. removed for anonymity] for network quality of service evolution, and constraints on the system state, from the authorizations and obligations which define policy decisions we gain clarity in the representation but also the ability to switch domain descriptions easily and study the behaviour of policies on different systems.

The choices we have made in the design of the language show that it is possible to encode subtle default relationships and decisions without sacrificing efficiency, readability or concision. The use of temporal constraints and an explicit representation of time has enabled us to express complex dependences of policy decisions on changing system states, as well as on other policies.

6. REFERENCES


domain-constrained policy. The consistency module takes as input a set \( \{F_1, \ldots, F_n\} \) of the literals to be checked for consistency, a (possibly empty) set \( C_{\text{loc}} \) of time constraints, a set \( \Delta \) of abducibles, and a set \( IC^* \) of dynamic constraints and \( P \).

\[ \text{Abduction}(G, C, \Delta, IC^*, II): \]
\[ \{ \text{returns a new } \Delta, \text{ new } C \text{ and new } IC^* \} \]
While \( G \) is not empty do
1. Get a literal \( L_i \) from \( G \)
2. If \( L_i \) is a positive atom with a non-abducible predicate, and there is a rule \( \phi, C_1 \rightarrow H \) in \( P \) where \( H, L_i \) unify with unifier \( \theta \), then
   \[ \text{Let } C = C \cup C_1; G = (G \setminus \{L_i\}) \cup \phi \theta; \]
3. If \( L_i \) is a literal with an abducible predicate and \( L_i \) unifies with an element in \( \Delta \) with unifier \( \theta \), then
   \[ \text{Let } G = (G \setminus \{L_i\}) \theta; \]
4. If \( L_i \) is an abducible positive literal that is not unifying with any element in \( \Delta \)
   \[ \text{Skolemize } L_i \text{ into } S_i \text{, and constraint } C' \]
   \[ \text{if } S_i \in \Delta \text{ then return failure} \]
   else
   \[ \text{Let } \Delta = \Delta \cup \{S_i\}; G = G \setminus \{L_i\}; C = C \cup C'; F = \text{Reduction}(IC^*, S_i); \]
   \[ \text{if } \text{Consistency}(F, C, \Delta, IC^*, II) \text{ returns } \Delta', C'' \quad \text{and } IC'' \]
   \[ \text{then } \]
   \[ \text{Let } C = C''; \Delta = \Delta'; IC'' = IC''; \]
   else return failure
5. If \( L_i \) is a non-abducible negative literal then
   \[ \text{Skolemize } L_i \text{ into } S_i \text{, and constraint } C' \]
   \[ \text{Let } C_{\text{loc}} = C \cup C'; \Delta = \Delta \cup \{S_i\}; \]
   \[ \text{if } \text{Consistency}(((\{S_i\}, \emptyset), C_{\text{loc}} \Delta, IC^*, II) \text{ returns } \Delta', C'' \quad \text{and } IC'' \]
   \[ \text{then } \]
   \[ \text{Let } C = C''; \Delta = \Delta'; IC'' = IC''; \]
   else return failure
6. If \( L_i \) does not match any of the previous cases then
   return failure
end while
return \( \Delta, C \) and \( IC^* \)
end Abduction

Figure 2: Abduction Procedure

The abduction module takes a literal \( L \) from the set passed as input and unfolds it in standard resolution fashion using the rules in \( P \), adding time constraints into a constraint store \( C \), until an abducible \( A \) is found. Whereas SLDNF would at this point fail the computation and backtrack, the abduction module treats the abducible \( A \) as a candidate hypothesis, and invokes the consistency module to see whether \( A \) can consistently be added to the current hypothesis \( \Delta \). The consistency check is important not only for the consistency of \( \Delta \) but also for the consistency of \( \Delta \cup P \). During abduction, negated non-abducible predicates are also added to \( \Delta \) (since no rule in \( P \) has negation in the head), requiring the consistency to check that \( P \cup \Delta \) does not prove their respective complements. Every consistency check has a separate branch of computation for each resolvent with \( P \) of the predicate to be checked for consistency. Every such re-
Consistency($F, C, \Delta, IC^*, \Pi$):
{returns a new $\Delta$, new C and new IC$^*$}

L: While $F$ is not empty do:
1. Select ($\{L_1, \ldots, L_n\}, C_{loc}$) from $F$ and let $F = F \setminus (\{L_1, \ldots, L_n\}, C_{loc})$;
2. If $C_{loc} \cup C$ is inconsistent GOTO L
3. Select either $C_{loc}$ or an $L_i$ from $\{L_1, \ldots, L_n\}$;
4. If an $L_i$ is selected and is an atom with no abducible predicate then
   For each $\phi \land C' \rightarrow H \in \Pi$ such that $H$ and $L_i$ unifies with unifier $\theta$ do
     if $\phi$ and $C'$ are empty and $n = i = 1$
       then return failure
     else Let $F = F \cup \{(\{L_1, \ldots, L_{i-1}, \phi, L_{i+1}, L_n\}, C \cup C')\theta\}$;
5. If an $L_i$ is selected and is a literal with an abducible predicate then
   For each $H \in \Delta$ such that $H = L_i\theta$ for some substitution $\theta$ do
     if $n = i = 1$ then return failure
   else
     Let $F = F \cup \{(\{L_1, \ldots, L_{i-1}, L_{i+1}, L_n\}, C)\theta\}$;
     Let $IC^* = IC^* \cup \{(\{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n\}, C)\}$
6. If $L_i$ is a negative literal with a not abducible predicate and it does not unify with any element in $\Delta$ then
   if Abduction($\{L^*_i\}, C, \Delta, IC^*, \Pi$) returns $\Delta', C''$ and $IC''$ then
     Let $\Delta = \Delta', C = C''$ and $IC^* = IC''$; else return failure;
7. If $C_{loc}$ is selected then find $C'$ such that $C \cup C'$ is consistent but $C \cup C' \cup C_{loc}$ is not and let $C = C \cup C'$
end while
return $\Delta, C$ and $IC^*$
end Consistency

Figure 3: Consistency Procedure

solvent is regarded as a proof that must be made finitely to fail for the consistency check to succeed. Failure of each resolvent occurs whenever at least one of its literals is made to fail. If needed, this failure can be explained by initiating a subordinate call of the abduction module in order to hypothesize some other abducibles (explicit or negated) to justify the failure. If all branches of the consistency call are passed (i.e. they fail) the calling abductive computation continues with the abducible $A$ added to $\Delta$ (along with any other abducibles accumulated during the consistency computation) and the constraint store $C$ (along with any other time constraints accumulated). If some branch of the consistency computation does not succeed (i.e. it cannot be made to fail) the calling abductive computation fails, indicating that $A$ is inconsistent with $\Delta \cup P$. In order to ensure consistency across its different branches of computation, the consistency module keeps track of constraints ($IC^*$) related to abducible predicates that unify with elements in $\Delta$. These can be seen as universally quantified assumptions about the abducibles in $\Delta$, generated during local consistency computations and which must hold for $\Delta$ to be an abductive explanation consistent with $P$. As the consistency computation can interleave with the abductive computation, the set $IC^*$ of dynamically generated constraints is also assumed to be a parameter of the abduction module.