Expressive Policy Analysis with Enhanced System Dynamicity

Arosha Bandara\(^3\), Seraphin Calo\(^2\), Robert Craven\(^1\), Jorge Lobo\(^2\), Emil Lupu\(^1\), Jiefei Ma\(^1\), Alessandra Russo\(^1\), and Morris Sloman\(^1\)

\(^1\) Department of Computing, Imperial College London
\(^2\) IBM, T.J. Watson Research Centre
\(^3\) The Open University, Milton Keynes

Abstract. Policy specification languages ought to include an analysis framework, which should at least \((i)\) be expressive, \((ii)\) be efficient, \((iii)\) take account of obligations and authorizations, \((iv)\) include a dynamic system model, and \((v)\) give useful diagnostic information. We present a logical policy analysis framework which has the satisfaction of these requirements at its heart, showing how significant policy-related properties can be analysed, and we give details of the existent implementation.

1 Introduction

There are many languages for the formal specification of authorization policies ([JSS97],[SYSR06]), some of which also provide some support for the modelling of obligations ([IYW06],[OAS05],[RDD07]) However, specification is only part of the story: to be useful, a system for policy engineering should also have associated with it an analysis framework. This lets the policies be checked for necessary and desirable properties. Some analysis frameworks exist ([IYW06],[BS04],[BN08]), but few of them take into account the changing system state, and allow the expression of temporal constraints and relationships amongst policy decisions. A policy framework should also allow the efficient evaluation of policies, and allow subtle dependencies of policies on a changeable system state to be expressed. Therefore, a model of the system is required as a component of an analysis framework, in order to determine whether policies conflict, and to analyse for properties that the system satisfies under a given set of policies. An analysis framework should also represent policies and the systems they regulate in a separable way—so that the behaviour of a policy on different systems, and the implementation of different policies in the same system, can easily be studied. This is essential given the common deployment of the same policy on heterogeneous systems. Policies which depend on other policies must also be expressible—whether or not someone is permitted to access a given resource may depend on whether they have fulfilled a relevant obligation.

This paper presents a logical framework for the specification and analysis of both authorization and obligation policies, which incorporates a model of the changing system state which satisfies all of the above requirements. We also provide an implementation, which gives policy engineers rich diagnostic information on analyses.
Using lessons learned from our work on the refinement of management policies [BLR+05], our broader aim is to develop a framework for policy refinement, from high-level general specifications, down to an abstract though implementable policy layer. The language we present here may be used to represent policies and systems at many different levels of abstraction and stages during the refinement process. Therefore the language must be expressive enough to represent policies at high and low levels. In this context, analysis is used to test whether a putative policy refinement satisfies constraints expressed at higher level of abstraction.

Our abstract logical language for policies is also expressive enough that different formalisms (e.g. Ponder2 [RDD07], XACML [OAS05], Cassandra [BS04]) can automatically be translated into it; in this way we make good the deficiencies of specification formalisms which have no analysis component. Translation algorithms have been developed for a large class of Ponder2 policies, and we are currently working on such schemes for XACML [OAS05] and Cassandra [BS04].

We can analyze for foundational, policy-focused principles, and application-specific properties, dependent on the particular system to which the policy applies, and that system’s structure. Specifically (though not exhaustively):

- **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 fulfillment.
- **Separation of duty** clashes, including static separation of duty, dynamic, and many other classes (see [SZ97] for terminology and instances).
- **Coverage gaps**, where no policy exists to dictate what the correct response to a request should be.
- **Policy comparison**, including the question of whether two policies are equivalent, or one is subsumed by the other.
- **Behavioural simulation**, where specific sequences of requests and events in the policy-regulated system are entered, to see the policy decisions which arise during the run.

In all cases, in the absence of a desired property our framework produces a complete list of the system traces which lead to the property’s failing, providing feedback on the conditions which result in system behaviour anomalies.

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 [LS01], 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 [JSSS01]. Defaults are also useful in the presence of conflicts: the policy combination rules of XACML [OAS05], 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.

We use abductive, constraint logic programming (ACLP) systems as the basis of our analysis algorithms and implementation, and the Event Calculus (EC) [KS86] to describe how events and actions occurring in the system affect
the system states. The EC allows us to specify how actions and events change system state, leading to circumstances in which a given policy rule is applicable. This information is an output of the analysis.

Others have also defined similar logic-based formalisms with well-defined complexity results for policy specification ([HW03], [JSSB97], [JSSS01]). Our approach, however, caters for more dynamic policy models where decisions may be based on temporal properties of the system. We have an explicit representation of time, with temporal variables governed by constraints, allowing policies to be highly sensitive to changing system state. In addition we include a class of obligation policies which effectively monitors when and how users or the system initiates actions. This is needed for managing security, but is also useful in other applications such as context-aware adaptation in ubiquitous computing and privacy. We have also developed an implementation, available at http://www.doc.ic.ac.uk/~rac101/ffpa/.

The paper is organized as follows. Section 2 considers related work. Section 3 gives the operational model and defines the basic terms and concepts used in our formal policy language. Section 4 gives the syntax and semantics of this language followed, including a number of illustrative examples. Section 5 discusses the kinds of analysis our language permits, together with a description of the kind of abduction we use. Conclusions and directions for future research are in Section 6.

2 Related Work

The work presented in this paper is based on our experience with formalizing policies using the Event Calculus [BLR03]. The Lithium language of Halpern and Weissman [HW03] is also a logical formalism; 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. As a simple example, assume there is a policy that all and only faculty members are permitted to chair committees; students are not [HW03]. Representing this requires a complete specification of faculty and student body members, which will need to be changed every time those members change, raising the well-known problem of elaboration tolerance [McC98]. 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 in our analyses 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 [DDLS01] and deontic logic; the result is similar to [IYW06]. 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 [IYW06].
Other formal languages take advantage of the computational efficiencies obtained by using subsets of logic programs, such as stratified logic. Barker presents in [Bar00] 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) [JSSB97], the Flexible Authorization Framework (FAF) [JSS97] and the extension to handle dynamic authorizations discussed in [CWJ04] are also based on stratified clausal-form logic. They offer techniques for detecting modality conflicts and some application-specific conflicts in authorization policies. However, they work with a fixed domain model; our framework caters for domain models which change as policy-regulated actions are performed.

In [BN07], 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 fall short of a fully dynamic model by limiting the specification to sequences of operations. Further, the classes of policy they expresses are in significant ways more limited than that supported by our language: 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 [BN08], the authors use abduction to analyse authorization policies, particularly focussing on finding explanations for the denial of access requests. They provide soundness, completeness and termination results. However, as with [BN07], 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 Language

Our operational model broadly follows the architecture and principle of operation of XACML [OAS05], PolicyMaker, and KeyNote [BFIK99]. 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.

We distinguish between regulatory predicates, used to describe the state of the PDP/PEP, and non-regulatory predicates that express the state of the policy-governed system. Regulatory predicates are subdivided into input regulatory, state regulatory, and output regulatory; similarly for non-regulatory predicates.
In general, a system moves between states depending on the occurrence of actions and events. Non-regulatory state predicates represent properties of states, and non-regulatory event predicates describe the occurrence of events. (The need for event predicates arises because, in general, not all occurrences which modify the state of a system are controllable by the policy mechanism.)

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. The policy representation language, $L^p$, includes at least the sorts $Subject$, $Target$, $Action$, the sets of $subjects$, $targets$ and $actions$, respectively, and the sort $Time$, given as $\mathbb{R}^+$, the set of non-negative reals, with constants including numerical constant symbols $(0, 1, \ldots)$ and variables such as $T$, with super- and subscripts as needed. Standard arithmetical functions $(+, -, /, \times)$ and relations ($=, \neq, <, \leq$ etc.) are presumed. Variables which range over $Subject$, $Target$, $Action$ are usually, respectively $Sub$, $Tar$, $Act$, again possibly with subscripts or superscripts. We insist on the possibility of representing the revocation of obligations, so that for all $Sub$, $Tar$, $Act \neq revoke(\cdots)$ and $T_s, T_e$, $revoke(Sub, Tar, Act, T_s, T_e) \in Action$. The predicates of $L^p$ are as defined in Table 1. For a formal definition of the language the reader is referred to [CLM⁺08].

The predicates $permitted$, $denied$ are self-explanatory. A particular instance $req(sub, tar, act, t)$ means that a request for $sub$ to perform $act$ on $tar$ is made at time $t$. Two instances $obl(sub, tar, act, t_s, t_e, t)$ and $fulfilled(sub, tar, act, t_s, t_e, t)$ (resp. $violated(sub, tar, act, t_s, t_e, t)$), denote that at time $t$, $sub$ is placed under an obligation to perform $act$ on $tar$ between $t_s$ and $t_e$, and that the obligation with these parameters has been fulfilled (resp. violated). Finally, an instance $cease_obl(sub, tar, act, t_{\text{init}}, t_s, t_e, t)$ is true at time $t$, if an obligation initially contracted by $sub$ at $t_{\text{init}}$ to perform $act$ on $tar$ between $t_s$ and $t_e$ is no longer binding. The language $L^p$ also includes an auxiliary predicate $reqInBetween$, whose instance $reqInBetween(sub, tar, act, t', t)$ represents that there is a request for $sub$ to perform $act$ on $tar$, at some time between $t'$ and $t$.

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 being regulated by the policy, the changes occurring both as a consequence of actions authorized and enforced by the PDP and PEP, and also as a result of events which are not under policy control. We use the Event Calculus [KS86] (EC) to model this
dynamicity in our domains. The language includes sorts Fluent (for representing dynamic features of states), Event (for system events not regulated by policies), Occurrence (for representing system events which are regulated by policies) and Time (as before). The predicates of $L^\text{EC}_D$, which model changing actions of the system and stem from the EC, are given in Table 2; the predicates of $L^\text{stat}_D$, representing unchanging properties of systems, are user-defined. For a formal definition of $L^D$, the reader is referred to [CLM+08].

<table>
<thead>
<tr>
<th>Event Calculus predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially($F$)</td>
</tr>
<tr>
<td>holdsAt($F$, $T$)</td>
</tr>
<tr>
<td>happens($E$, $T$)</td>
</tr>
<tr>
<td>broken($F$, $T_1$, $T_2$)</td>
</tr>
<tr>
<td>initiates($E$, $F$, $T$)</td>
</tr>
<tr>
<td>initiates($Occ$, $F$, $T$)</td>
</tr>
<tr>
<td>terminates($E$, $F$, $T$)</td>
</tr>
<tr>
<td>terminates($Occ$, $F$, $T$)</td>
</tr>
</tbody>
</table>

Table 2. Domain description language $L^\text{EC}_D$: the predicates

The policy analysis language $\mathcal{L} = \mathcal{L}^\pi \cup \mathcal{L}^D$, is the union of the policy representation and domain description languages, with the sort Occurrence given by the terms $\text{Sub}, \text{Tar}, \text{Act}$ for every $\text{Sub} \in \text{Subject}$, $\text{Tar} \in \text{Target}$, and $\text{Act} \in \text{Action}$.

4 Axioms and Policies

4.1 Authorizations

We first define time constraints, and then the structure of authorization rules.

**Definition 1.** A time constraint $C$ is an expression of the form $\tau_1 \rho \tau_2$, where each $\tau_i$ is a constant or variable of type Time, or an arithmetic linear expression built using $+, -, \times, \div$, Time constants and variables, and where $\rho$ is one of $=, \neq, <, \leq, >, \geq$. \hfill ⊥

**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 or atoms preceded by the negation-by-failure not, taken from the set $\mathcal{L}^\pi \cup \mathcal{L}^\text{stat}_D \cup \{\text{holdsAt, happens, broken}\}$; the $C_i$ are time constraints;  
2. any variable appearing in a time constraint must also appear somewhere other than in a time constraint;  
3. $\text{Sub}, \text{Tar}, \text{Act}, T$ are terms of type Subject, Target, Action and Time respectively;
4. for the time argument $T_i$ of each $L_i \notin \mathcal{L}_{stat}$, 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 a regulatory output predicate; if $L_i \in \mathcal{L}_{EC}$, then it should either be $\text{holdsAt}$ 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 will be imposed later.)

Condition 1, on the predicates which may appear in the body of our authorization policy rules, excludes the \textit{initiates} and \textit{terminates} predicates which are used to express the laws of system evolution. Condition 4, on the time arguments of our predicates, is imposed to ensure that authorization does not depend on 'future' properties.

\textbf{Example 1.} “Alice may delete classified data files from her device if she sends a notification to the supplier of the data 10 minutes in advance, and the supplier does not respond to the notification asking Alice to retain the file.”

We represent this as follows:

\[
\begin{align*}
\text{permitted}(\text{alice, device, delete}(F), T) & \leftarrow \\
\text{holdsAt}(\text{fileDesc}(S, F, \text{class}), T_n), \\
\text{do}(\text{alice, S, notify(delete(F)), T_n}), T = T_n + 10, \\
\text{not reqInBetween}(S, F, \text{retain}(F), T_n, T).
\end{align*}
\]

The predicate \texttt{reqInBetween} is related to the operator SINCE of temporal logics [Gol92]; the utility of having such a predicate has been demonstrated to us repeatedly in many formalizations of policy rules. To capture its semantics, the following rule is always included in our framework:

\[
\begin{align*}
\text{reqInBetween}(\text{Sub, Tar, Act}, T') & \leftarrow \text{req}(\text{Sub, Tar, Act}, T_r), T' \leq T_r \leq T.
\end{align*}
\]

An instance \texttt{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.

When gathering together authorization rules to form an authorization policy, it is normal to include a number of other, more general rules. Some of these are used to specify the behaviour of the PEP in response to the PDP—such as 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 the behaviour is in the presence or absence of a policy rule stating that the request is to be denied, and so on. We see it as a virtue of our framework for policy analysis 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 many policy systems in heterogeneous environments. Consider the three example availability rules in Table 3. The basic availability rule is
more stringent: according to it, an action is enforced by the PEP only when it has been positively permitted by the PDP—similar to [LS01]. The positive availability rule is less strict: it enforces the performance of an action as long as that action has not been expressly denied by the policy rules. The negative availability rule describes one possible response to the denial of a request for action; we imagine that the main function of the regulatory output predicate deny will be for auditing purposes, for instance to record when an action which was denied had to be overridden and permitted in an emergency.

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

Many more policy regulation rules are possible than those given as examples in Table 3; all are optional inclusions in an authorization policy.

**Definition 4.** An authorization policy is a set \( \Pi \) of authorization rules, together with the req-in-between rule, and a set of policy regulation rules, such that \( \Pi \) is locally stratified.\(^4\)

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:

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

### 4.2 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 [IYW06]. As in most deontic logics, obligations may be fulfilled or not, allowing us to represent the behaviour of systems of which humans are a part. Our approach to modelling obligations is similar to [IYW06].

\(^4\) A 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 computationally hard but large classes of rules can be identified easily based on the time index [NRG05].
Definition 5. An obligation policy rule is a formula

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

where the conditions 1–4 as for Definition 2 hold, with the addition that \( T_s \) and \( T_e \) should be variables of type Time. (That \( t_s < t_e \) is not syntactically required, but sensible 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:

\[ \text{fulfilled} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e, T) \leftarrow \]
\[ \text{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e, T_{\text{init}}), \text{do} (\text{Sub}, \text{Tar}, \text{Act}, T'), \]
\[ \text{not } \text{cease}_{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e, T'), T_{\text{init}} \leq T_s \leq T' < T_e, T' < T. \]

\[ \text{violated} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e, T) \leftarrow \]
\[ \text{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e, T_{\text{init}}), \text{not } \text{cease}_{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_{\text{init}}, T_s, T_e, T'),
\[ T_{\text{init}} \leq T_s < T_e \leq T. \]

An obligation is fulfilled when the action a subject has been obliged to perform is executed (notice that the \text{do} in the body of the rule here means that the execution of such an action must first be authorized by the system). An obligation is violated when no such action occurs. The rules for \text{fulfilled} and \text{violated} use \text{cease}_{obl} as a subsidiary predicate, defined by the following rules:

\[ \text{cease}_{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_{\text{init}}, T_s, T_e, T) \leftarrow \]
\[ \text{do} (\text{Sub}, \text{Tar}, \text{Act}, T'), T_s \leq T' < T \leq T_e. \]
\[ \text{cease}_{obl} (\text{Sub}, \text{Tar}, \text{Act}, T_{\text{init}}, T_s, T_e, T) \leftarrow \]
\[ \text{do} (\text{Sub}', \text{Sub}, \text{revoke} (\text{Sub}, \text{Tar}, \text{Act}, T_s, T_e), T'), T_{\text{init}} \leq T' < T \leq T_e. \]

\text{cease}_{obl} is a state regulatory predicate used to mark the fact that something has occurred which would cause an obligation to cease. This can happen if the obligation is fulfilled or revoked; there are therefore two clauses defining \text{cease}_{obl}. The \text{cease}_{obl} rule for revocation makes use of the \text{revoke} members of the sort \text{Action}, introduced in Section 3; 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 \text{revoke} argument identify the obligation to be revoked.

Example 2. “A connecting node must re-indentify 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 might be partially formalized as follows:

\[
\text{obl}(U, \text{serv}, \text{sub}_2\text{ID}(U, \text{serv}), T + 0.1, T + 300, T + 0.1) \leftarrow \\
\text{holdsAt(node}(U), T \rangle, \text{do}(U, \text{serv}, \text{connect}(U, \text{serv}), T) \\
\text{obl}(	ext{serv}, \text{serv}, \text{disconnect}(U, \text{serv}), T_e, T_{e+1}, T_e) \leftarrow \\
\text{violated}(U, \text{serv}, \text{sub}_2\text{ID}(U, \text{serv}), T_s, T_e, T_e).
\]

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 after the server connects to the node—in the rule, we have assumed there is a delay of 0.1 seconds, but in practice this interval can be made as small as possible without being equal to zero, reflecting the limitations of the system clock.

**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 \( \Pi = \Pi_a \cup \Pi_o \) is any union of an authorization policy \( \Pi_a \) and an obligation policy \( \Pi_o \).

### 4.3 Event Calculus and 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 logic-based formalism, where the 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. Core axioms are then added, common to any EC formalization, to relate the behaviour specifications of the \text{initiates} and \text{terminates} axioms to the holding or not of state properties. The EC is used to model, analyse and implement many dynamic systems (see [MS02] for references). Details of the core axioms are given in Appendix A.

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 either 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 \( \mathcal{L}_{\text{stat}}^D \): an initiates axiom and a 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 L^D_{EC}.

Each \( L_i \) is a literal of an atom in \( L^{D}_{stat} \), 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.

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 mixed policy \( \Pi \) and a domain description \( D \).

(Note that if \( \Pi \) and \( D \) are locally stratified, then so is \( \Pi \cup D \).)

### 4.4 Separation of Duty and Chinese Wall Policies

Separation of duty (SoD) [SZ97] and Chinese Wall policies [BN89] 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. For example: a person cannot advise company A’s transactions once he has advised company B’s, and vice versa. This can be represented as the following negative authorization rules:

\begin{align*}
\text{denied}(Sub, b, \text{advise}, T) \leftarrow & \text{do}(Sub, a, \text{advise}, T'), T' < T. \\
\text{denied}(Sub, a, \text{advise}, T) \leftarrow & \text{do}(Sub, b, \text{advise}, T'), T' < T.
\end{align*}

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

\begin{align*}
\text{denied}(\text{Subject}, \text{roles}, \text{activate}(\text{role}_a), T) \leftarrow & \text{holdsAt}(\text{isActivated}(\text{Subject}, \text{role}_b), T). \\
\text{denied}(\text{Subject}, \text{roles}, \text{activate}(\text{role}_b), T) \leftarrow & \text{holdsAt}(\text{isActivated}(\text{Subject}, \text{role}_a), T).
\end{align*}

Other classes of SoD policy can be captured similarly.

### 4.5 Semantics

We use the standard stable model semantics [GL88] 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 \( \{ \text{initially}, \text{happens} \} \cup L^{D}_{stat} \) and any set \( \Delta^\pi \) of ground instances of the regulatory predicate \( \text{req} \). The sets \( \Delta^D \) and \( \Delta^\pi \) 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 $\Delta^D$, $\Delta^\pi$ 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($P \cup \Delta^D \cup \Delta^\pi$). We let model($P \cup \Delta^D \cup \Delta^\pi$) refer to the (unique) stable model of ground($P \cup \Delta^D \cup \Delta^\pi$).

5 **Policy Analysis**

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), can be converted into the task of seeing whether (stable) models of the domain-constrained policy verify a number of properties. For instance, we may wish to prove

$$\forall T (\neg(\text{permitted}(\text{sub}, \text{tar}, \text{act}, T) \land \text{denied}(\text{sub}, \text{tar}, \text{act}, T)))$$

for ground terms sub, tar, act. If this property is provable, then all well and good. If not, 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 $\Delta^\pi$ (as described in Section 4.5) such that the property is not true, i.e. whether

$$\text{model}(P \cup \Delta^D \cup \Delta^\pi) \models \exists T (\text{permitted}(\text{sub}, \text{tar}, \text{act}, T) \land \text{denied}(\text{sub}, \text{tar}, \text{act}, T))$$

This is equivalent to showing that the previous formula is false, and can be solved using *Abductive Logic Programming* (with constraints—ACLP), which computes the sets $\Delta^D$ and $\Delta^\pi$. The output to 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. We currently use an abductive constraint logic programming proof procedure based on that found in [KMM00] to implement our system. (For the details of the algorithm, see Appendix B.)

Following the same schema, we can define formulas to check for other properties. For example, violations of dynamic SoD can be checked with:

$$\text{model}(P \cup \Delta^D \cup \Delta^\pi) \models \exists T (\text{permitted}(\text{sub}, \text{roles}, \text{activate}(\text{role}_a), T) \land \text{permitted}(\text{sub}, \text{roles}, \text{activate}(\text{role}_b), T))$$

5 Where $X$ is a set of formulas, $\text{ground}(X)$ is the set of ground instances of members of $X$. 

12
Also, coverage analysis can be performed using:

\[
    \text{model}(P \cup \Delta^D \cup \Delta^\pi) \models \exists \text{Sub}, \text{Tar}, \text{Act}, T(\text{req}(\text{Sub}, \text{Tar}, \text{Act}, T) \\
    \land \neg \text{permitted}(\text{Sub}, \text{Tar}, \text{Act}, T) \\
    \land \neg \text{denied}(\text{Sub}, \text{Tar}, \text{Act}, T))
\]

This query will find instances of requests for which there is no associated permission or denial according to the authorization rules of the policy. Other properties can be handled in similar ways.

Further examples and system traces can be found on the website for the implementation at \url{http://www.doc.ic.ac.uk/~rac101/ffpa/}.

5.1 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.

We insist that 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 mixed policy component \( \Pi \), there is a maximum value \( t \) such that whenever a body of a policy rule is made true by the model, all time indices must belong to some interval \([t_s, t_s + t]\), and if we also insist that 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

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

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 dependencies amongst the members of our mixed policies (see Definition 7). We do this by insisting that there is a total ordering amongst the triples \((\text{Sub}, \text{Tar}, \text{Act})\), such that whenever an authorization or obligation policy rule contains \( \text{Sub}, \text{Tar}, \text{Act} \) in the head with time index \( T \), all literals with time index \( T' = T \) in the body of the predicates \( \text{permitted}, \text{denied}, \text{obl} \) can only contain \( \text{Sub}', \text{Tar}', \text{Act}' \) such that \((\text{Sub}', \text{Tar}', \text{Act}') < (\text{Sub}, \text{Tar}, \text{Act})\) in the ordering. Further, we also insist that whenever a negative literal in the body of a policy rule contains a variable, that variable should also appear in some positive literal of the body. (This way we ensure that selection of literals during policy evaluation is safe in the sense of logic-programming.)
**Result 1.** Under these conditions, a result from [Cho95] can be used to show that the evaluation of queries for predicates of permitted, denied and obl can be performed in time polynomial in the length of the preceding history relevant to queries, these histories being bounded by the size of the language that we assume to be finite.

Authorizations are typically evaluated when a req is received for permission to perform an action; the fulfillment of obligations can be monitored using techniques such as view maintenance in relational databases or a version of the RETE algorithm for production rules.

**Result 2.** In the case of the analysis tasks using the ACLP abductive procedures, the presence of the total ordering we have imposed on (Sub, Tar, Act) triples (together with the other constraints imposed above) can be used to show that our abduction analyses always terminate.

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 [SYSR06] and in [IYW06] 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.

### 5.2 Implementation

A prototype implementation of our formal analysis framework is freely available to download from [http://www.doc.ic.ac.uk/~rac101/ffpa/](http://www.doc.ic.ac.uk/~rac101/ffpa/). The implementation uses the open-source abductive constraint logic programming ASYSTEM [VN04]. Tests have enabled us to find modality conflicts, coverage problems, and other interesting properties of policies in conjunction with system descriptions.

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. To this end we are currently looking at alternatives to the ASYSTEM as a basis for our implementation.

### 6 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 expressivity 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.

Our formalization separates the representation of the laws of system evolution, and constraints on the system state, from the authorizations and obligations which define policy decisions. The gains in clarity of representation this has afforded, together with the ability to switch domain descriptions easily and study the behaviour of policies on different systems, is a convincing vindication of our approach. 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 dependencies of policy decisions on changing system states, as well as on other policies.

Abductive Constraint Logic Programming is a very 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 heterogenous 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.

We are continuing work on the implementation. At the moment, all suitable ACLP systems use integers as a basis of their constraint, but the modularity of the abductive approach we have taken means that an implementation based on reals—as formalized in our framework—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 [RDD07], and are working on schemes for XACML [OAS05] and others.

We also have a representation of obligations which, instead of having explicit temporal arguments, are constrained by the occurrence of events. Space limitations have prevented us from presenting this work in the current paper.

We have also started work on a refinement framework, of which the analysis framework in the current paper will form a part. Our previous work on the theme of policy refinement [BLR+05] for network quality of service management suggests that many of the properties we have built into our analysis framework (expressivity, separation of the laws for system change from policies, flexible expression of defaults, etc.) should also be present in formalisms for policy refinement.
References


The Event Calculus

The EC is a well-studied formalism, variants of which exist both as logic programs and in first-order logical axioms (using a second-order axiom to enforce a circumscriptive semantics), with the ability concisely to represent the effect of actions on properties of a system, and built-in support for the default persistence of fluents.

The set $EC$ of Event Calculus core axioms is given in Table 4. The first clause specifies that a changeable property of the system holds at time $T$, if
that property held at time 0 and nothing disturbed its default persistence. The next two clauses 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 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 [KS86], or for recent approaches, [MS02].

Table 4. Event Calculus Core Axioms

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>holdsAt(F, T) ← initially(F), not broken(F, 0, T).</td>
</tr>
<tr>
<td>holdsAt(F, T) ← initiates(Sub: Tar: Act, F, T), T_s &lt; T,</td>
</tr>
<tr>
<td>do(Sub, Tar, Act, T_s), not broken(F, T_s, T).</td>
</tr>
<tr>
<td>holdsAt(F, T) ← initiates(Event, F, T_s), T_s &lt; T,</td>
</tr>
<tr>
<td>happens(Event, T_s), not broken(F, T_s, T).</td>
</tr>
<tr>
<td>broken(F, T_s, T) ← terminates(Sub: Tar: Act, F, T'), T_s &lt; T &lt; T'.</td>
</tr>
<tr>
<td>do(Sub, Tar, Act, T'), T_s &lt; T' &lt; T.</td>
</tr>
<tr>
<td>broken(F, T_s, T) ← terminates(Event, F, T'),</td>
</tr>
<tr>
<td>happens(Event, T'), T_s &lt; T' &lt; T.</td>
</tr>
</tbody>
</table>

B Abduction and Consistency Algorithm Modules

We here show details of the abduction algorithm which forms the heart of our framework. It is composed Abductive and Consistency modules.\(^6\)

The abductive proof procedure is composed of two modules, the abduction phase and consistency phase, which interleave. The former is based on SLDNF, and takes as input a set of literals \(L_1, \ldots, L_n\), a (possibly empty) set \(C\) of time constraints (also called the constraint store), a set \(\Delta\) of abducibles, a set \(IC^*\) of dynamically-generated constraints, and \(P\), the 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 \(C_{loc}\) of time constraints, a set \(\Delta\) of abducibles, and a set \(IC^*\) of dynamic constraints and \(P\).

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.

\(^6\) Note that we define Reduction\((IC^*, L) = \{([L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n}, C) \theta \mid ([L_1, \ldots, L_n], C) \in IC^* \text{ and } L_\theta = L\theta\)
Abduction($G, C, \Delta, IC^*, \Pi$):

\{returns a new $\Delta$, new $C$ 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 $\Pi$ where $H, L_i$ unify with unifier $\theta$, then

   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

   Let $G = (G \setminus \{L_i\})\theta$;

4. If $L_i$ is a literal with an abducible predicate not unifying with any element in $\Delta$

   Skolemize $L_i$ into $S_i$, and constraint $C'$

   if $S_i \in \Delta$ then return failure

   else

   Let $\Delta = \Delta \cup \{S_i\}; G = G \setminus \{L_i\}; C = C \cup C'; F = \text{Reduction}(IC^*, S_i)$;

   if Consistency($F, C, \Delta, IC^*, \Pi$) returns $\Delta', C''$ and $IC^*_1$ then

   Let $C = C''; \Delta = \Delta'; IC^* = IC^*_1$;

   else return failure

5. If $L_i$ is a non-abducible negative literal then

   Skolemize $L_i$ into $S_i$, and constraint $C'$

   Let $C_{loc} = C \cup C'; \Delta = \Delta \cup \{S_i\}$;

   if Consistency($\{\{S_i\}, \emptyset\}, C_{loc}, \Delta, IC^*, \Pi$) returns $\Delta', C''$ and $IC^*_1$ then

   Let $C = C''; G = G \setminus \{L_i\}; \Delta = \Delta'; IC^* = IC^*_1$;

   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

Fig. 1. Abduction Procedure

with $P$ of the predicate to be checked for consistency. Every such resolvent 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
Consistency($F, C, \Delta, IC^*, II$):
\{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 $\phi \land C' \rightarrow H \in II$ 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($\{L_i^*\}, C, \Delta, IC^*, II$) 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

Fig. 2. Consistency Procedure

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.