Expressive Policy Analysis with Enhanced System Dynamicity*

Robert Craven
Department of Computing
Imperial College, London
robert.craven@imperial.ac.uk

Jorge Lobo
IBM T.J. Watson Research Center
jlobo@us.ibm.com

Jiefei Ma
Department of Computing
Imperial College, London
jm103@doc.ic.ac.uk

Alessandra Russo
Department of Computing
Imperial College, London
ar3@doc.ic.ac.uk

Emil Lupu
Department of Computing
Imperial College, London
e.c.lupu@imperial.ac.uk

Arosha Bandara
Department of Computing
Open University
a.k.bandara@open.ac.uk

ABSTRACT

Despite several research studies, 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.

Categories and Subject Descriptors

K.6.4 [**Computing Milieux**]: Management of Computing and Information Systems—*System Management*; K.6.1 [**Computing Milieux**]: Management of Computing and Information Systems—*Project and People Management*

General Terms

Design, Management

Keywords

Policies, Formal analysis, Security, Authorization

*Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

ASIACCS '09, March 10–12, 2009, Sydney, NSW, Australia

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 needs several key properties. First, it should be *expressive*. We need to specify both authorization [23, 32] and obligation policies [21, 30, 31] formally, and to allow complex dependencies of one on the other—for instance, authorization to withdraw a library book may be denied if the obligation to return requested items has been repeatedly violated. 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 [26], for example, has blanket denials for actions not covered explicitly by policy rules. Whilst we 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 [22]. Defaults are also useful in the presence of conflicts: the policy combination rules of XACML [30], 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—[21, 7, 6]—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 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 insufficient. Policy authors also need to check for a wide variety of properties on the policies and systems they define. This requires an expressiveness in the query language of the analysis component, and strength in the analysis algorithm itself. Analysis should be able to cope with the following 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 [33] for terminology and instances).
- **Coverage gaps**, where no policy exists to dictate the correct response to a request.
- **Policy comparison**, including whether two policies are equivalent, or one is subsumed by the other.
- **Behavioural simulation** provides specific sequences of requests and events to the policy-regulated system to determine which policy decisions arise.

Third, the output of the analysis component should be rich enough to provide useful diagnostic information to a policy writer or system engineer. The system traces, policy decisions, and actions related to a queried property should all be indicated by the analysis—in addition to properties of the actors or principals involved, and the policy rules used in making the decisions that led to the state reached. Alternative traces and decisions should also be suggested, and the user ought to be able to insert constraints which guide the search for diagnostic information.

Fourth, it is desirable that a policy framework is separated into a part which is used to describe the policies, and a part which specifies system behaviour—by which we mean, the system on which the policies are deployed, and whose requests and actions the policies govern and shape [14]. This permits analyzing the behaviour of policies on different systems—something which is crucial, given the increasing deployment of the same policy on heterogeneous platforms—and spares the user the effort of formalizing the policy again for each new system.

This paper presents an expressive logical framework for policy specification, in which it is possible to analyse for the types of properties mentioned above. The framework caters for both authorization and obligation policies, and incorporates a model of the changing system state. Our algorithm gives policy authors rich diagnostic information on analyses. We use abductive, constraint logic programming (ACLP) systems as the basis of our analysis algorithms and implementation, and the Event Calculus (EC) [25] to describe how events and actions occurring in the system affect the system states, leading to circumstances in which a given policy rule is applicable. This information is an output of the analysis.

In contrast to other logic-based formalisms ([20, 24, 22]), our approach caters for more dynamic policy models and includes an explicit representation of time, with temporal variables governed by constraints, allowing policies to be sensitive to changing system state. In addition, we include a class of obligation policies which monitors when and how users or the system initiate actions. This is needed for managing security, but is also useful in other applications such as context-aware adaptation in ubiquitous computing and privacy. Our policy language is also expressive enough that existing policy notations such as Ponder2 [31], XACML [30] and Cassandra [7] can automatically be translated into it; to provide analysis for specification notations which currently do not support this. Translation algorithms have been developed for a large class of Ponder2 policies, and we are currently working on such schemes for XACML. We have developed a prototype implementation,[1] of our analysis framework.

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 future research are in Section 5.

2. RELATED WORK

The Lithium language [20] 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* [20] requires a complete specification of faculty and student body members, which may change dynamically, thus raising the well-known problem of *elaboration tolerance* [27]. The use of default rules—of the kind that our formalism supports—can simplify specifications and changes to the specification. 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 experience with Ponder [13] and deontic logic; the result is similar to [21]. However, we have adapted obligation policies to produce a more general language that allows more complex policies to be represented, and our framework can support analyses such as the *strong accountability* checking presented in [21]. Dougherty et al. [15] present a model in which obligations are tied to authorizations, as conditions on acquiring permission to access a given resource. The model we use is more general, allowing obligations that are not tied to authorizations, as well as mutual dependence. [15] also includes a system model, though this is conceived abstractly as a set of state traces, which would need to be defined in

[1] `http://www.doc.ic.ac.uk/~rac101/ffpa/`

full—our use of EC domain descriptions allows us to generate traces of actions which lead to the holding or violation of policy-related properties, from concise system descriptions.

Barker presents in [4] a language that represents 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) [24], the Flexible Authorization Framework (FAF) [23] and the extension to handle dynamic authorizations discussed in [11] 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; in contrast, our framework does not presume a predefined domain model and can cater for varying system descriptions.

An access control policy language is presented in [5]; it has an associated analysis framework based on a subset of transaction logic programs and 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, 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 [6], the authors use abduction to analyse authorization policies, focussing on finding explanations for the denial of access requests. They provide soundness, completeness and termination results. However, as with [5], there is no fully dynamic system model. So 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.

Fisler et al. [17] present an approach for the analysis of role-based access control policies written in XACML. The underlying formalism is MTBDDs (multi-terminal binary decision diagrams), and the method supports the analysis of the impact of changes on policies. There are, however, a number of limitations. Obligation policies are not supported, and there is no scope for policies that simultaneously permit and deny a given access request (a direct and oft-noted consequence of the nature of XACML's combination rules). We regard support, especially of the latter, as essential for any generic policy analysis tool. The query language (for expressing properties) is somewhat cumbersome, though the authors acknowledge this as a theme for future work. Further, though environmental constraints can be included, there is no explicit dynamic system model, and so analysis does not show system traces and policy decisions which lead to properties of interest.

Finally, in [9, 10] the authors define a simple but powerful framework for representing and reasoning about access-control policy composition. The semantics for access requests is four-valued: permit, deny, undefined, and conflict. Analysis is performed by transforming properties to be checked into constraints which can be fed to a model-checker; the approach can support coverage-gap, modality conflict, and policy comparison analysis. The emphasis of this work is on the definition of an expressive, generic semantics for policy composition and related analysis, and thus there is no system model—the system governed by a policy is treated as a 'black box'. By contrast, our two-pronged approach (policies and systems) is intended to enable the output of diagnositic information about which system traces can give rise to which policy properties. We also consider a wider class of policies, including obligation policies.

Input regulatory	**Output regulatory**
$req(Sub, Tar, Act, T)$	$do(Sub, Tar, Act, T)$
	$deny(Sub, Tar, Act, T)$
State Regulatory	
$permitted(Sub, Tar, Act, T)$	$denied(Sub, Tar, Act, T)$
$obl(Sub, Tar, Act, T_s, T_e, T)$	
$fulfilled(Sub, Tar, Act, T_s, T_e, T)$	
$violated(Sub, Tar, Act, T_s, T_e, T)$	
$cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T)$	

Table 1: Policy analysis language $\mathcal{L}^{\pi}$: the predicates

3. POLICIES

3.1 Preliminaries

Our operational model broadly follows the architecture and operation of XACML [30]. 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 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*, $\mathcal{L}^{\pi}$, 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 ($=$, $\neq$, $<$, $\leqslant$ etc.) are presumed. The predicates of $\mathcal{L}^{\pi}$, which we call *regulatory predicates*, are shown in Table 1. 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. Instances $obl(sub, tar, act, t_s, t_e, t)$ and $fulfilled(sub, tar, act, t_s, t_e, t)$ (or $violated(sub, tar, act, t_s, t_e, t)$), 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, a given instance $cease\_obl(sub, tar, act, t_{init}, t_s, t_e, t)$ is true at time t, if an obligation initially contracted by *sub* at t_{init} to perform *act* on *tar* between t_s and t_e is no longer binding. Our language models the revocation of obligations, so we include $revoke(Sub, Tar, Act, T_s, T_e) \in Action$, for each action *Act* which is not itself a revocation, to allow this. The *output regulatory* predicate *do* records whether an action is allowed

to occur by the PEP—flattening the essentially four-valued logic of our policy language into the required two values—and *deny* is included for auditing purposes; the logic of these is discussed in Section 3.2.

The *domain description language* $\mathcal{L}^D = \mathcal{L}^D_{EC} \cup \mathcal{L}^D_{stat}$, is used to represent both changing and unchanging properties of the system regulated by the policy. We use the Event Calculus [25] (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 $\mathcal{L}^D_{EC}$ are standard in the EC (see Section 3.4); the predicates of $\mathcal{L}^D_{stat}$, for unchanging properties of systems, are user-defined.

A complete definition of the language is in [2].

3.2 Authorizations

Definition 1 *A* time constraint C *is an expression of the form* $\tau_1\ \rho\ \tau_2$*, where each* τ_i *is a constant or variable of type Time, or an arithmetic linear expression built using* $+, -, /, *$*, Time constants and variables, and where* ρ *is one of* $=, \neq, <, \leqslant, >, \geqslant$. ⌟

Note that 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*

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

1. *the* L_i *are atoms taken from the set*

$$\mathcal{L}^\pi \cup \mathcal{L}^D_{stat} \cup \{holdsAt, happens, broken\}^2$$

possibly preceded by the negation-by-failure **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;*

3. *Sub, Tar, Act, T are terms of type Subject, Target, Action and Time respectively;*

4. *for the time argument* T_i *of each* $L_i \notin \mathcal{L}^D_{stat}$*, we must have* $C_1 \wedge \cdots \wedge C_n \models T_i \leqslant T$*;*[3] *if* $C_1 \wedge \cdots \wedge C_n \models T_i = T$ *then the* L_i *must not be an output regulatory predicate and if in addition* $L_i \in \mathcal{L}^D_{EC}$*, 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 will be imposed later.)* ⌟

Condition 4 is necessary to ensure that authorizations do not depend on 'future' properties.

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."* ⌟

[2]For the meaning of these predicates, see Section 3.4.

[3]We use $\models$ standardly, as FOL semantic entailment.

We represent this as follows:

$$\begin{aligned} permitted(N, D, delete, T) \leftarrow\ & holdsAt(fileDesc(D, class), T_n),\ T = T_n + 10, \\ & holdsAt(owner(D, O), T_n), \\ & do(N, O, notify(delete, D), T_n), \\ & \textbf{not}\ reqInBetween(O, N, retain(D), T_n, T). \end{aligned}$$

The predicate *reqInBetween* is related to the operator SINCE of temporal logics [19]; we have found such a predicate useful on several occasions. To capture its semantics, the following rule is always included in our framework:

$$reqInBetween(Sub, Tar, Act, T', T) \leftarrow req(Sub, Tar, Act, T_r), T' \leqslant T_r \leqslant T.$$

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) [33] 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 follows:

$$\begin{aligned} denied(Sub, M_1, assist, T) \leftarrow\ & do(Sub, M_2, assist, T'),\ T' < T, \\ & holdsAt(activity\_type(M_1, medical), T), \\ & holdsAt(activity\_type(M_2, survey(A)), T'), \\ & holdsAt(area\_classify(A, contaminated), T'). \end{aligned}$$

Simple dynamic SoD policies that define mutually exclusive role activation can be handled as follows:

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

Similar encodings can be done for other classes of SoD policy.

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 [26]. The positive

$do(Sub, Tar, Act, T) \leftarrow$ $\quad req(Sub, Tar, Act, T), permitted(Sub, Tar, Act, T).$	Basic availability
$do(Sub, Tar, Act, T) \leftarrow$ $\quad req(Sub, Tar, Act, T), \textbf{not } denied(Sub, Tar, Act, T).$	Positive availability
$deny(Sub, Tar, Act, T) \leftarrow$ $\quad req(Sub, Tar, Act, T), denied(Sub, Tar, Act, T).$	Negative availability

Table 2: Policy Regulation Rules

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 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* Π *of authorization rules, with the definition of reqInBetween, and policy regulation rules, such that* Π *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:

$$permitted(Sub, Tar, Act, T) \leftarrow$$
$$\quad group(Sub, root), \textbf{not } denied(Sub, Tar, Act, T).$$

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 [21]. 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*

$$obl(Sub, Tar, Act, T_s, T_e, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

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. (*$T_s < T_e$ *is not required, but sensible obligation policy rules will always include constraints which make this true.)* ⌟

[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, in general, computationally hard; but large classes of rules can be identified as locally stratified easily based on the time index [29].

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

$$fulfilled(Sub, Tar, Act, T_s, T_e, T) \leftarrow$$
$$\quad obl(Sub, Tar, Act, T_s, T_e, T_{init}),\ do(Sub, Tar, Act, T'),$$
$$\quad \textbf{not } cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T'),$$
$$\quad T_{init} \leqslant T_s \leqslant T' < T_e,\ T' < T. \tag{1}$$

$$violated(Sub, Tar, Act, T_s, T_e, T) \leftarrow$$
$$\quad obl(Sub, Tar, Act, T_s, T_e, T_{init}), \tag{2}$$
$$\quad \textbf{not } cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T_e),$$
$$\quad T_{init} \leqslant T_s < T_e \leqslant T.$$

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 action must be allowed by the authorization policies). 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:

$$cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T) \leftarrow$$
$$\quad do(Sub, Tar, Act, T'), T_s \leqslant T' < T \leqslant T_e.$$

$$cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T) \leftarrow$$
$$\quad do(Sub', Sub, revoke(Sub, Tar, Act, T_s, T_e), T'),$$
$$\quad T_{init} \leqslant T' < T \leqslant T_e.$$

cease_obl denotes 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 and may also be constrained by authorization policies. The parameters of the *revoke* argument identify the obligation to be revoked.

Example 3 *"A connecting node should re-indentify itself within five minutes of making a connection to a server, or 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 obli-

gation. They can be formalized as follows:

$$obl(U, serv, sub2ID(U, serv), T{+}\epsilon, T{+}300, T{+}\epsilon) \leftarrow$$
$$holdsAt(node(U), T),\ do(U, serv, connect(U, serv), T).$$

$$obl(serv, serv, disconnect(U, serv), T_e, T_e{+}1, T_e) \leftarrow$$
$$violated(U, serv, sub2ID(U, serv), T_s, T_e, T_e).$$

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

Definition 6 *An* obligation policy Π *is a set of obligation rules, with the 'fulfilment', 'violation' and 'cease obl' rules, such that* Π *is locally stratified.* ⌟

Definition 7 *A* security policy $\Pi = \Pi_a \cup \Pi_o$ *is any union of an authorization policy* Π_a *and an obligation policy* Π_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 circumscriptive semantic). 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, analyse and implement many dynamic systems (see [1] for a recent example, or [28] for general references).

In the EC, effects of events or occurrences are defined by two predicates *initiates* and *terminates*. *initiates* describes which state properties are caused hold due to an event; and *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 *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 *initiates* and *terminates* axioms to state properties. These core axioms, the set *EC*, are shown below.

$$holdsAt(F, T) \leftarrow$$
$$initially(F),\ \textbf{not}\ broken(F, 0, T). \tag{3}$$

$$holdsAt(F, T) \leftarrow$$
$$initiates(Sub{:}Tar{:}Act, F, T_s),\ T_s < T,$$
$$do(Sub, Tar, Act, T_s),\ \textbf{not}\ broken(F, T_s, T). \tag{4}$$

$$holdsAt(F, T) \leftarrow$$
$$initiates(Event, F, T_s),\ T_s < T,$$
$$happens(Event, T_s),\ \textbf{not}\ broken(F, T_s, T). \tag{5}$$

$$broken(F, T_s, T) \leftarrow$$
$$terminates(Sub{:}Tar{:}Act, F, T'),$$
$$do(Sub, Tar, Act, T'),\ T_s < T' < T. \tag{6}$$

$$broken(F, T_s, T) \leftarrow$$
$$terminates(Event, F, T'),$$
$$happens(Event, T'),\ T_s < T' < T. \tag{7}$$

The first clause (3) 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 (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 [25], or for recent approaches, [28].

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 $\mathcal{L}^D_{stat}$. 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 $D = EC \cup D'$ *contains the core axioms 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* $\mathcal{L}^D_{stat}$*; a* state constraint

$$holdsAt(F, T) \leftarrow L_1, \ldots, L_n.$$

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

$$initiates(X, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$
$$terminates(X, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

such that:

- $initiates(X, F, T), terminates(X, F, T) \in \mathcal{L}^D_{EC}$.
- *Each* L_i *is a literal of an atom in* $\mathcal{L}^D_{stat}$*, or else a literal of the predicate 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* $C_1 \wedge \cdots \wedge C_n \models T_i \leqslant T$.

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 [16]. We represent *user-to-role assignment* by the fluent $hasRole(Subject, Role)$ and *permission-to-role assignment* by $hasPerm(Role, Resource, Action)$. The access con-

trol can then be expressed by the following axiom:

$$permitted(Sub, Resource, Act, T) \leftarrow$$
$$holdsAt(hasRole(Sub, Role), T),$$
$$holdsAt(hasPerm(Role, Resource, Act), T).$$

The following axioms capture the role hierarchy inheritance:

$$holdsAt(subrole(R, R'), T) \leftarrow$$
$$holdsAt(contains(R', R), T).$$

$$holdsAt(subrole(R, R'), T) \leftarrow$$
$$holdsAt(contains(R'', R), T),$$
$$holdsAt(subrole(R'', R'), T).$$

$$holdsAt(hasRole(U, R), T) \leftarrow$$
$$holdsAt(hasUser(R, U), T).$$

$$holdsAt(hasRole(U, R), T) \leftarrow$$
$$holdsAt(hasUser(R', U), T),$$
$$holdsAt(subrole(R', R), T).$$

One advantage of the EC is that using the same formalism we can also express the administration of RBAC (ARBAC—see e.g. [32]). First, we use EC rules to model the effects of user-role assignments (i.e. adding or removing assignments):

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

Then we model the effects of the role-permission assignments:

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

And finally, operations to the role hierachy:

$$initiates(Admin{:}R{:}addRole(R'), contains(R, R'), T).$$
$$terminates(Admin{:}R{:}removeRole(R'), contains(R, R'), T).$$

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,

$$initially(hasUser(admin, alice)).$$
$$initially(hasPerm(admin, R, assignUser(U))).$$
$$initially(hasPerm(admin, R, unassignUser(U))).$$
$$initially(hasPerm(admin, R, assignPerm(T, A))).$$
$$initially(hasPerm(admin, R, unassignPerm(T, A))).$$
$$initially(hasPerm(admin, R, addRole(R'))).$$
$$initially(hasPerm(admin, R, removeRole(R'))).$$

We now show a system trace. In general, these are sequences of actions, 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.

$$do(alice{:}medical\_aid{:}addRole(field\_surgeon, 0)$$
$$do(alice{:}medical\_aid{:}assignPerm(S, initialExamine), 1)$$
$$do(alice{:}field\_surgeon{:}assignUser(daneeka), 2)$$
$$do(alice{:}field\_surgeon{:}assignPerm(S, operate), 3)$$
$$do(alice{:}medical\_aid{:}assignUser(duckett), 4)$$

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 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 SOD in this RBAC model, between users assiged to the roles of *Medical Aid* and *Security Officer*.

$$denied(Admin, sec\_officer, assignUser(U), T) \leftarrow$$
$$holdsAt(hasUser(medical\_aid, U), T).$$

$$denied(Admin, medical\_aid, assignUser(U), T) \leftarrow$$
$$holdsAt(hasUser(sec\_officer, U), T).$$

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-constrainted 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* Π *and a domain description* D*, such that* P *is locally stratified.* ⌟

We use the standard *stable model* semantics [18] of logic programs. To capture the operational model we start with any set Δ^D of ground instances of non-regulatory predicates from the set $\{initially, happens\} \cup \mathcal{L}^D_{stat}$ and any set Δ^π of ground instances of the regulatory predicate *req*. The sets Δ^D and Δ^π 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 Δ^D, Δ^π 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* $ground(P \cup \Delta^D \cup \Delta^\pi)$.[5]

[5]Where X is a set of formulas, $ground(X)$ is the set of ground instances of members of X.

We let model($P \cup \Delta^D \cup \Delta^\pi$) refer to the (unique) stable model of ground($P \cup \Delta^D \cup \Delta^\pi$). ⌟

4. POLICY ANALYSIS

4.1 Types and Examples

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

(*i*) *Modality Conflicts.* A task of analysing a domain-constrained policy $P = \Pi \cup D$ to see 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 from a particular kind of modality conflict:

$$\forall T(\neg(permitted(sub, tar, act, T) \wedge denied(sub, tar, act, T))) \quad (8)$$

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 is the task of checking whether there are inputs Δ^D and Δ^π (see Section 3.5) such that the property is *not* true, i.e. whether:

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

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 Δ^D and Δ^π. 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 found in [34].

(*ii*) *Illustration of trace abduction.* One of the scenarios we have been using concerns a natural disaster rescue, 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]

$$denied(Sub, Tar, move(L), T) \leftarrow holdsAt(is\_injured(Tar, spinal), T). \quad (9)$$

[Medics are allowed to move a patient with a spinal injury if they are on a spine board.]

$$permitted(M, Tar, move(L), T) \leftarrow holdsAt(is\_injured(Tar, spinal), T), \; holdsAt(on\_spine\_board(Tar), T). \quad (10)$$

[Injuries who are located in a house at risk of collapse must be moved to hospital within 10 mins by a medic]

$$obl(M, Tar, move(hosp), T_s, T_e, T_s) \leftarrow T_e = T_s + 10, \; holdsAt(at(Tar, H), T_s), \; holdsAt(at\_risk(H), T_s), \; happens(find(X, Tar), T_s), \; holdsAt(is\_injured(Tar, InjuryType), T_s). \quad (11)$$

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 fulfil, because of the presence of a conflicting authorization policy? There are several different interpretations of this query, one of which is the following:

$$obl(Sub, Tar, Act, T_s, T_e, T_{init}) \wedge \textbf{not } cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T) \wedge denied(Sub, Tar, Act, T) \wedge T_s < T. \quad (12)$$

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).

$$obl(medic, alice, move(hosp), 1, 1, 11) \wedge \textbf{not } cease\_obl(medic, alice, move(hosp), 1, 1, 11, 2), \wedge denied(medic, alice, move(hosp), 2) \wedge 1 < 2. \quad (13)$$

The abduced atoms, the union of the sets Δ^D and Δ^π, were:

$$\{ initially(at\_risk(house3)), \; initially(at(alice, street)), \; happens(walk(alice, house3), 0), \; happens(injure(alice, spinal), 1) \} \quad (14)$$

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 her 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 rule (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. Our previous experience developing policy analysis tools has shown that it is possible to build interfaces that hide much of the formal representations, presenting information in a format less expert users can understand.

(*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

$$happens(injure(alice, spinal), 1)$$

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:

$$\begin{aligned} &obl(Sub, Tar, Act, T_s, T_e, T_{init}) \\ &\wedge \textbf{not } cease\_obl(Sub, Tar, Act, T_{init}, T_s, T_e, T) \\ &\wedge denied(Sub, Tar, Act, T) \wedge T_s < T \\ &\wedge \textbf{not } holdsAt(injure(alice, spinal), T) \end{aligned} \tag{15}$$

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

$$\begin{aligned} \{\, &initially(at\_risk(house3)),\ initially(at(bob, street)), \\ &happens(walk(bob, house3), 0), \\ &happens(injure(bob, spinal), 1)\,\} \end{aligned} \tag{16}$$

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

$$\begin{aligned} \{\, &initially(at\_risk(house3)),\ initially(at(bob, street)), \\ &initially(at(alice, street)),\ initially(injured(bob, leg\_break)), \\ &happens(carry(alice, bob, house3), 0), \\ &happens(injure(bob, spinal), 1)\,\} \end{aligned} \tag{17}$$

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* 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:

$$\begin{aligned} &model(P \cup \Delta^D \cup \Delta^\pi) \models \\ &\quad \exists T(permitted(sub, roles, activate(role\_a), T) \\ &\qquad \wedge permitted(sub, roles, activate(role\_b), T)) \end{aligned}$$

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 in which a request for action is not governed by policy decisions. 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:

$$\begin{aligned} &model(P \cup \Delta^D \cup \Delta^\pi) \models \\ &\quad \exists Sub, Tar, Act, T(req(Sub, Tar, Act, T) \\ &\qquad \wedge \neg permitted(Sub, Tar, Act, T) \\ &\qquad \wedge \neg denied(Sub, Tar, Act, T)) \end{aligned}$$

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:

$$\begin{aligned} &req(Sub, Tar, triage, T) \wedge \textbf{not } permitted(Sub, Tar, triage, T) \\ &\wedge \textbf{not } denied(Sub, Tar, triage, T) \\ &\wedge \textbf{not } holdsAt(is\_injured(Tar, spinal), T) \end{aligned} \tag{18}$$

finds many answers, depending on the number of subjects and targets in the domain.

This kind of coverage 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 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:

$$do(Sub, Tar, Act, T) \leftarrow \textbf{not } denied(Sub, Tar, Act, T).$$

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:

$$\begin{aligned} &req(Sub, Tar, Act, T) \wedge \textbf{not } permitted(Sub, Tar, Act, T) \\ &\quad \wedge do(Sub, Tar, Act, T). \end{aligned}$$

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 for 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

$$\textbf{not } holdsAt(is\_injured(Tar, spinal), T)$$

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:

$$permitted(duckett, P, initialExamine, 2).$$

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.* In this form of analysis, we check to see whether one policy is included in another, whether one implies another, or they are equivalent, etc.

Our analysis framework allows us to test for these inclusions, enabling an engineer who modifies a policy to prove whether his modifications would have an effect, and whether any added elements are, in fact, redundant. Suppose, for example, that the current policy set is $\{(9), (10), (11)\}$, as in our running example, and let us say that the proposed new positive authorization rule is

[Patients in category 'z' are allowed to be moved]

$$permitted(M, Tar, move(L), T) \leftarrow holdsAt(category(Tar, z), T). \quad (19)$$

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

$$initiates(injure(Tar, spinal), is\_injured(Tar, spinal), T).$$
$$initiates(injure(Tar, spinal), category(Tar, z), T).$$
$$terminates(cure(Tar, spinal), is\_injured(Tar, spinal), T).$$
$$terminates(cure(Tar, spinal), category(Tar, z), T).$$

In this case, adding the rule (19) to the policy would have no effect. If Π_2 denotes the policy set $\{(9), (10), (11)\}$, and Π_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. ($\Pi_1 \subseteq_D \Pi_2$ means that, given a domain description D, for all system traces, permissions, denials, or obligations implied by the policy Π_1 are also implied by the policy Π_2.)

One way of checking whether $\Pi_1 \subseteq_D \Pi_2$ is to relativize the policy representation languages, so that the state regulatory predicates receive a subscript. Policies from Π_1 would then be written using $permitted_1$, obl_1, and so on; and policies from Π_2 would be written using $permitted_2$, obl_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:

$$permitted_1(Sub, Tar, Act, T) \wedge \textbf{not } permitted_2(Sub, Tar, Act, T),$$

$$denied_1(Sub, Tar, Act, T) \wedge \textbf{not } denied_2(Sub, Tar, Act, T),$$

$$obl_1(Sub, Tar, Act, T_s, T_e, T) \wedge \textbf{not } obl_2(Sub, Tar, Act, T_s, T_e, T)$$

each returned no answers, given a domain description D, this would be a proof that $\Pi_1 \subseteq_D \Pi_2$.

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 Π_1 extends the obligations of users on the medical team to move patients, compared to policy Π_2, by a query such as:

$$obl_1(Sub, Tar, move(L), T_s, T_e, T) \wedge \textbf{not } obl_2(Sub, Tar, move(L), T_s, T_e, T) \wedge \textbf{not } holdsAt(team(Sub, medical), T)$$

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

4.2 Implementation

A prototype implementation of our formal analysis framework is freely available to download.[6] The implementation uses the open-source abductive constraint logic programming ASystem [34]. 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 examined, analysis results under these modifications would hold under the original version of the axioms with $\mathbb{R}$. However, the systems 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.

4.3 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 SLDNF.

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 Π, 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 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

$$permitted(Sub, Tar, Act, T) \leftarrow holdsAt(f, T'),\ T = T' + 10.$$

is in the policy, we know we must record information about whether the fluent f holds 10 seconds in the past; beyond 10

[6] From `http://www.doc.ic.ac.uk/~rac101/ffpa/`.

seconds, we may not care (depending on the other policies in Π) 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 policies (see Definition 7). We do this by insisting that there is a total ordering amongst the triples (*Sub*, *Tar*, *Act*), such that whenever an authorization or obligation policy rule contains *Sub*, *Tar*, *Act* in the head with time index T, all literals with time index $T' = T$ in the body of the predicates *permitted*, *denied*, *obl* can only contain *Sub*′, *Tar*′, *Act*′ such that (*Sub*′, *Tar*′, *Act*′) < (*Sub*, *Tar*, *Act*) in the ordering. Further, 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.)

Under these conditions, a result from [12] can be used to show that the evaluation of queries for literals 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. We also have soundess and completeness theorems for our formal framework, for policy evaluation queries.

In the case of the analysis tasks using the ACLP abductive procedures, matters are more complicated. We have 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 $[t_s, t_s + t]$ 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 recurse 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 [32] and in [21] 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 implementation of abduction we use now is more general than strictly required in our analyses.

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 expresiveness 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 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.

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 [31], and are working on schemes for XACML [30] and others.

Our broader objective is to define a refinement framework, of which the analysis framework 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 [3] 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.) are also valuable for policy refinement.

6. ADDITIONAL AUTHORS

Additional authors: Seraphin Calo (IBM T.J. Watson Research Center, email: `scalo@us.ibm.com`) and Morris Sloman (Department of Computing, Imperial College London, email: `m.sloman@doc.ic.ac.uk`).

7. REFERENCES

[1] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Extracting requirements from scenarios with ilp. In S. Muggleton, R. P. Otero, and A. Tamaddoni-Nezhad, editors, *ILP*, volume 4455 of *LNCS*, pages 64–78. Springer, 2006.

[2] A. Bandara, S. Calo, R. Craven, J. Lobo, E. Lupu, J. Ma, A. Russo, and M. Sloman. An expressive policy analysis framework with enhanced system dynamicity. Technical Report, Department of Computing, Imperial College London, 2008.

[3] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou. Policy refinement for diffserv quality of service management. In *Integrated Network Management*, pages 469–482. IEEE, 2005.

[4] S. Barker. Security policy specification in logic. In *Proc. of Int. Conf. on AI*, pages 143–148, June 2000.

[5] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. In *ESORICS*, pages 203–218, 2007.

[6] M. Y. Becker and S. Nanz. The role of abduction in declarative authorization policies. In P. Hudak and D. S. Warren, editors, *PADL*, volume 4902 of *LNCS*, pages 84–99. Springer, 2008.

[7] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *CSFW*, pages 139–154. IEEE Computer Society, 2004.

[8] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on S & P*, pages 206–214, 1989.

[9] G. Bruns, D. S. Dantas, and M. Huth. A simple and expressive semantic framework for policy composition in access control. In P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, editors, *FMSE*, pages 12–21. ACM, 2007.

[10] G. Bruns and M. Huth. Access-control policies via belnap logic: Effective and efficient composition and analysis. In *CSF*, pages 163–176. IEEE Computer Society, 2008.

[11] S. Chen, D. Wijesekera, and S. Jajodia. Incorporating dynamic constraints in the flexible authorization framework. In *ESORICS*, pages 1–16, 2004.

[12] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[13] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In M. Sloman, J. Lobo, and E. Lupu, editors, *POLICY*, volume 1995 of *LNCS*, pages 18–38. Springer, 2001.

[14] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 632–646. Springer, 2006.

[15] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In *ESORICS*, pages 375–389, 2007.

[16] D. Ferraiolo and D. Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, 1992.

[17] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 196–205. ACM, 2005.

[18] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, Washington, August 15-19 1988.

[19] R. Goldblatt. *Logics of time and computation*. Center for the Study of Language and Information, Stanford, CA, USA, 2nd edition, 1992.

[20] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4), 2008.

[21] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proc. of ACM CCS*, pages 134–143, 2006.

[22] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[23] S. Jajodia, P. Samarati, and V. Subrahmanian. A logical language for expressing authorizations. In *Proc. of the IEEE Symposium on S & P*, pages 31–42, 1997.

[24] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. of the ACM SIGMOD Conf.*, May 1997.

[25] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[26] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[27] J. McCarthy. Elaboration tolerance. In *Proc. Common Sense 98*, 1998.

[28] R. Miller and M. Shanahan. Some alternative formulations of the event calculus. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *LNCS*, pages 452–490. Springer, 2002.

[29] C. Nomikos, P. Rondogiannis, and M. Gergatsoulis. Temporal stratification tests for linear and branching-time deductive databases. *Theor. Comput. Sci.*, 342(2-3):382–415, 2005.

[30] OASIS XACML TC. extensible access control markup language (XACML) v2.0, 2005.

[31] G. Rusello, C. Dong, and N. Dulay. Authorisation and conflict resolution for hierarchical domains. In *Proc. of IEEE Policy Workshop*, June 2007.

[32] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[33] R. Simon and M. E. Zurko. Separation of duty in role-based environments. In *CSFW*, pages 183–194. IEEE Computer Society, 1997.

[34] B. Van Nuffelen. *Abductive constraint logic programming: implementation and applications*. PhD thesis, K.U.Leuven, Belgium, June 2004.