A Formal Framework for Policy Analysis

Robert Craven,[1] Jorge Lobo,[2] Emil Lupu,[1] Jiefei Ma,[1]
Alessandra Russo,[1] Morris Sloman,[1] Arosha Bandara[3]

1. Imperial College, London; 2. IBM, T.J. Watson Research Centre;
3. Open University, Milton Keynes.

Abstract

We present a formal, logical framework for the representation and analysis of an expressive class of authorization and obligation policies. Basic concepts of the language and operational model are given, and details of the representation are defined, with an attention to how different classes of policies can be written in our framework. We show how complex dependencies amonst policy rules can be represented, and illustrate how the formalization of policies is joined to a dynamic depiction of system behaviour. Algorithmically, we use a species of abductive, constraint logic programming to analyse for the holding of a number of interesting properties of policies (coverage, modality conflict, equivalence of policies, etc.). We describe one implementation of our ideas, and conclude with remarks on related work and future research.

1 Introduction

This paper presents a formal framework for the representation and analysis of authorization and obligation policies. Policies are represented in an expressive logical language, which can capture complex dependencies amongst policy rules; these policy rules are then joined to a specification of the behaviour of the system which the policies regulate, and many different types of analyisis and verification may then be performed on the joined system.

The policies themselves may be originally specified by a user in a language such as Ponder2 [RDD07] [1] or XACML [OAS05], and then translated by machine into our formalism. For the purposes of this paper we pass over the details of the original specification of policies, and the process of machine translation into our own formalism, and concentrate on the core language which is used for analysis, giving illustrations of the forms of policy rule our language can represent, and showing examples of the typical kinds of analysis we make.

A crucial component of our framework is the representation, of dynamic system behaviour (we have been influenced in seeing the necessity for this by [DFK06]). We use the Event Calculus [KS86] to describe how events and actions occurring in the system effect the system states; these specifications of system behaviour may be made at whichever level of abstraction the user chooses. The rules defining system behaviour are then joined to a simple model of policy enforcement, together with concrete authorization and obligation policies, in order to provide a complete, analyzable, and highly dynamic model of policy-constrained systems. Perhaps the most obvious benefit of this type of approach is the possibility of searching for configurations of the policy mechanisms which are dependent on dynamically-generated

[1] See also `http://ponder2.net/`.

configurations of system state—such as the presence of a modality conflict (e.g. an action's being simultaneously permitted and denied) only after a given sequence of system events. Consider, for example, a policy which states that the primary health-carer of a patient is permitted to access the patient's record, but that nurses in training are denied such access. The two policy rules given have a potential conflict: for if nurses in training may also be primary health-carers, then such a nurse would both be permitted and denied access to his or her patient's records. However, the medical system in question may be engineered in such a way as to prevent, not as a consequence of policy rules but as a result of the behaviour of the system, nurses in training from becoming primary carers—and thus the potential modality conflict will not arise. It is only by analysing policies in conjunction with the laws of system evolution that a reliable analysis is obtained.

Our formalism is based on a subset of first-order logic which may be written as the clauses of a program in a logic-programming language such as PROLOG, with a semantic difference in the treatment of negation; we describe a current implementation of our ideas, and give the results of some experimental analyses. As a consequence of the correspondence between our framework and logic programming, we can give a standard logic-programming semantics to the axioms defining system and policy models. Further, the correspondence with logic programming allows us to use abductive, constraint logic programming (ACLP) systems as the basis of our analysis algorithms, and offers an advantage of expressivity over approaches which use temporal logic operators [TN96].

Others have also defined similar logic based formalisms with well-defined complexity results for policy specification; our work here has partly been inspired by such notable examples as [HW03] and [JSSB97, JSSS01]. Our approach, as mentioned, caters for more dynamic policy models where policy decisions may be based on temporal properties of the system. 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.

In summary, we feel the main contributions of our framework, and the language and algorithms at its heart, are related to three main concepts. *Dynamicity*—our policies depend on changing features of systems, and these changes are represented and modelled within the formalism to allow us to reason about them, and the policy configurations to which they give rise. *Expressivity*—we have used an expressive language based on normal logic programs, allowing us to depict and reason about a large number of authorization and obligation policies, and their interactions (for examples, see later in the paper). *Hard analysis tasks*—our analysis algorithms are able to cope with a large number of analysis problems in the context of reasoning about policies, including interesting classes of modality conlict and coverage analysis (again, for the details see later in the paper).

The paper is organized as follows. Section 2 introduces the operational model behind policy enforcement. Section 3 defines the basic terms and concepts used in our formal policy language, and Section 4 gives the syntax and semantics of this language followed, in Section 5, by examples to demonstrate its appropriateness and expressiveness. Section 6 discusses the kinds of analysis our language permits, together with a description of the kind of abduction we use. Section 7 presents some related work on formalizing security policies, and in Section 8 we offer conclusions and directions for future research.

2 Operational Model

We wish to reason about and analyze the interaction of policies and systems, where policies will be used to regulate some of the behaviour of the system. In doing this, we will need to understand both how policies are used and enforced in the system, and also how the system itself evolves—what the laws are which determine its evolution. In general, then, we follow the architectural structure of policy systems such PolicyMaker and KeyNote [BFIK99], and XACML [OAS05]. We will separate out two components of the worlds we model—the policy decision and enforcement mechanisms, and the systems to which those policies refer, and which they modify.

A *policy decision point* (or PDP) will have access to a repository of policies. The policies may be of several different types; in the current paper, we focus on *authorization policies* and *obligation policies*.

In the case of authorization policies, inputs to the PDP take the form of requests for a given *subject* to perform a specified *action* on a *target*; the PDP then decides whether or not the action in question should be authorized or denied, using *authorization policies* which exist in the repository. In coming to a decision about whether to allow or disallow an action, the PDP will, in general, need to be aware both of other permissions and obligations which currently hold (so that decisions may recurse over policies); and what we will call 'non-regulatory' facts about the system, facts which are not expressed as permissions, denials, or obligations, such as the size in bytes of a file. If the PDP determines that a given request to perform an action should be authorized, then it notifies a *policy enforcement point* (PEP), which then executes the authorized action, on the specified *target*, within the system.

The policy repository will also contain obligation policies according to which a given *subject* is bound to perform an *action* on a *target*—at the moment, between two times T_s and T_e (though we are considering other ways of delimiting the period during which an action ought to be performed). If the conditions associated with an obligation policy are satisfied during the time interval specified, the PDP will inform the PEP of the action to be performed. The PDP is also responsible for notifying the PEP that an obligation has been fulfilled or violated (i.e. the time during which it could have been fulfilled has passed without the fulfilling action), and the PEP can enforce an appropriate action. One virtue of our language is that it leaves unspecified what that action should be—any remedial or censorial action will be defined by the policy itself. (This approach of allowing a great deal of flexibility in how policies are implemented, is something we see as a strength of our language.)

The outline of the role of the PDP and PEP above suggests that, in introducing a language to represent and reason about policies and their interactions with the systems whose behaviour they regulate, we will have at the most general level two kinds of predicate. *Regulatory* predicates will describe the state of the PDP and PEP, including inputs and outputs to both, and *non-regulatory* predicates will represent the current state of the system whose behaviour is being governed by policies. The PDP/PEP may receive inputs (in the form of requests for permission to perform actions, possibly with certificates of authentication to verify the identity of the requester), and it also issues outputs (which represent the decision to execute a permitted action). Further, the PDP has an internal state, representing which actions are permitted, and which subjects are constrained by obligations, at any given time. We thus subdivide the regulatory predicates of our language into *input regulatory*, *state regulatory*, and *output regulatory* predicates.

Similar divisions will also be made in the category of *non-regulatory* predicates,

which describe the systems whose behaviour is being governed by policies. In general, a system is conceived to be in a given state, and to move between states, depending on the performance of actions and occurrence of events. We introduce a category of *non-regulatory state predicates* to represent properties of states, and *non-regulatory event predicates* to 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 and vetted by the policy mechanism attached to the system.)

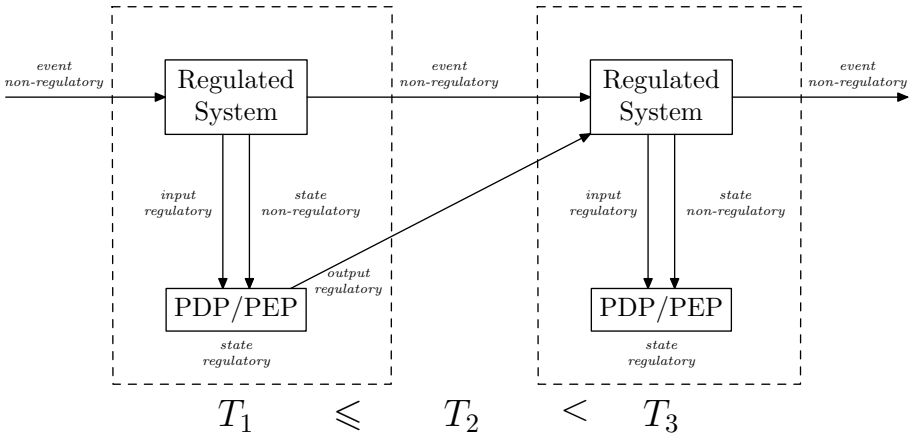

Figure 1: Operational Model used in our framework

The behaviour outlined above is represented in Figure 1. In the diagram, the workings of the system at two different logical time points, T_1 and T_3, are represented. The diagram also shows the occurrence of actions at a third logical time T_2, such that $T_1 \leqslant T_2 < T_3$; it is presumed that this is the only time-point *between* T_1 and T_3 at which actions occur.

3 Language

We use many-sorted first-order predicate logic as our base language, although it should be borne in mind that when giving the semantics later (in Section 4.5), negation (which we write as **not**) will be interpreted, logic-programmatically, as negation-by-failure. We assume conventional concepts including that of an *atom*, and a literal is an atom or a negated atom (not necessarily grounded).

Throughout the paper, constants, functions, and predicate symbols begin with a lower-case letter, and variables begin with an upper-case letter. The symbol $\vec{X}$ will be used to denote a tuple of variables of possibly different sorts, the symbol $\vec{x}$ to denote a tuple of ground terms of possibly different sorts. The symbols $\top$ and $\bot$ are 0-ary predicates receiving the standard interpretation.

We have a sort *Time*, given as $\mathbb{R}^+$, the set of non-negative reals, with constants including numerals for integers $(0, 1, \ldots)$ and variables such as T, with super- and subscripts as needed. Standard arithmetical functions ($+$, $-$, $/$, $*$) and relations ($=$, $\neq$, $<$, $\leqslant$ etc.) are presumed.

Further sorts we include are *Subject*, *Target*, *Action*, the sets of *subjects*, *targets* and *actions*, respectively. Variables which range over these sets are usually,

respectively, *Sub*, *Tar*, *Act*, again possibly with subscripts or superscripts. Where variable-naming conventions differ from those we set out here, this is mentioned. We insist on the inclusion of a member $revoke(Sub, Tar, Act, T_s, T_e) \in Action$, where $Act \in Action - \{revoke(\cdots)\}$, which will be given special treatment in the axioms defining system behaviour we present in Section 4; this action represents the revocation of an obligation.

We divide our language into components, the syntactic divisions marking off different functions for the predicates contained within them.

Definition 1 A *policy analysis language* $\mathcal{L}^\pi = \mathcal{L}^\pi_i \cup \mathcal{L}^\pi_s \cup \mathcal{L}^\pi_o$ is a many-sorted, first-order logic language, with (at least) the sorts *Subject*, *Target*, *Action* and *Time*, with

$$\{revoke(Sub, Tar, Act, T_s, T_e) \mid Sub \in Subject, Tar \in Target, \\ T_s, T_e \in Time, Act \neq revoke(\cdots)\} \subseteq Action$$

whose predicates are partioned into the following sets.

- A set $\mathcal{L}^\pi_i$ of *input regulatory predicates*, containing only the 4-place predicate *req*, whose arguments must be, in order, of sorts *Subject*, *Target*, *Action* and *Time*; a grounded instance $req(sub, tar, act, t)$ represents that a request for *sub* to perform *act* on *tar* was notified to the PDP at time t.
- A set $\mathcal{L}^\pi_s$ of *state regulatory predicates*, containing the following predicates:
  - *permitted* (4-place, arguments $Subject \times Target \times Action \times Time$); an atom $permitted(sub, tar, act, t)$ is used to represent that *sub* has the permission to perform *act* upon the target *tar* at time t.
  - *denied* (4-place, arguments $Subject \times Target \times Action \times Time$); a ground instance $denied(sub, tar, act, t)$ represents that at time t, *sub* is denied authorization to perform *act* on *tar*.
  - *obl* (6-place, arguments $Subject \times Target \times Action \times Time \times Time \times Time$); a ground instance $obl(sub, tar, act, t_s, t_e, t)$ represents that at time t, *sub* is placed under an obligation to perform *act* on *tar* between t_s and t_e.
  - *fulfilled* (6-place, arguments $Subject \times Target \times Action \times Time \times Time \times Time$); a ground instance $fulfilled(sub, tar, act, t_1, t_2, t)$ means that at time t, it is true that *sub* fulfilled its obligation to perform *act* on *tar* between t_1 and t_2.
  - *violated* (6-place, arguments $Subject \times Target \times Action \times Time \times Time \times Time$); a ground instance $violated(sub, tar, act, t_s, t_e, t)$ means that at time t, it is true that *sub* violated (i.e., failed to fulfil) its obligation to perform *act* on *tar* between t_s and t_e.
  - *cease_obl* (7-place, arguments $Subject \times Target \times Action \times Time \times Time \times Time \times Time$); a ground instance $cease\_obl(sub, tar, act, t_{init}, t_s, t_e, t)$ is true at 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. (For the purposes of intuitive clarification, we state now that this occurs either (i) when the obligation has been fulfilled, and t is after the time of its fulfilment, or (ii) when the obligation was revoked before it was fulfilled or violated).

– *reqInBetween* (5-place predicate, arguments $Subject \times Target \times Action \times Time \times Time$); an instance $reqInBetween(sub, tar, act, t', t)$ represents that there was a request for *sub* to perform *act* on *tar*, at some time between t' and t.

- A set $\mathcal{L}_o^\pi$ of *output regulatory predicates*, containing the predicates
  - *do* (4-place, arguments $Subject \times Target \times Action \times Time$); a grounded instance $do(sub, tar, act, t)$ is used to express that *sub* performs the action *act* on *tar* at time t.
  - *deny* (4-place, arguments $Subject \times Target \times Action \times Time$); a grounded instance $deny(sub, tar, act, t)$ is used to express that *sub* is refused permission to perform the action *act* on *tar* at time t, subsequent to a request.⌟

Each time we model an application which is controlled by a PDP, for the purpose of analysing possible conflicts in the policy rules and proving other properties about the interaction of policies and systems, we specify a policy analysis language which conforms to the definition above—which means specifying what membership of the sorts *Subject*, *Target* and *Action* is: the range of actors within the systems, and the actions they can perform on targets. The predicates from $\mathcal{L}^\pi$ will be used in representing the effects of authorization and obligation policy rules, as described in Section 4. Other sorts may also be needed, for example for quantification over arguments to members of *Action*—these additional sorts are dependent on the specific application.

In order to reason about the interactions of policies with systems which they regulate, and also to identify possible lacks of coverage of a security policy, or to check whether two policies are equivalent, we need to yoke our representations of policies to a model of system behaviour, which tells us how a system evolves over time as a consequence of actions or events occurring in it. One way to see why a system model is necessary, is to consider the task of analysing a policy for the presence of *modality conflicts*. We may wish to know whether a given policy simultaneously authorizes and denies a certain action—something which the expressivity of our formalism allows. Yet circumstances under which given policy rules both permit and deny a given request may, though seemingly possible from an examination only of those rules, in fact never arise in practice owing to physical constraints on the evolution of the system.

Accordingly, we introduce the other main component of the languages we will use in reasoning about policies. This is used to represent both changing and unchanging properties of the system being regulated by the policy, where the changes can occur 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 the control of policies. We will use a common variant of the Event Calculus [KS86] to model this dynamicity in our domains; axioms are presented later—for now, we present only the predicates and sorts of the language.

We have the sorts *Fluent* (for representing dynamic features of states), *Event* (for representing system events not regulated by policies) *Action* (for representing system events which *are* regulated by policies) and *Time* (as before).

Definition 2 A *domain description language* $\mathcal{L}^D = \mathcal{L}^D_{EC} \cup \mathcal{L}^D_{stat}$ is a many sorted, first-order language with (at least) the sorts *Fluent*, *Event*, *Occurrence* and *Time*. The *Time* sort is $\mathbb{R}^+$, as before. Predicates are partitioned into sets as follows.

- A set $\mathcal{L}^D_{EC}$ of (non-regulatory) *event calculus predicates*, containing:
  - *initially* (1-place, argument *Fluent*), with a specific instance *initially*(f) representing that the non-regulatory state property represented by f holds at time 0 (i.e., in the initial state of the system).
  - *holdsAt* (2-place, arguments *Fluent* $\times$ *Time*), where a specific instance *holdsAt*(f, t) represents that the non-regulatory state property represented by f holds at time t.
  - *happens* (2-place, arguments *Event* $\times$ *Time*), where a particular atom *happens*(e, t) represents that the non-regulated event e occurs at time t.
  - *broken* (3-place, arguments *Fluent* $\times$ *Time* $\times$ *Time*); a grounded instance *broken*(f, t_s, t) represents that the fluent f ceased to hold at some time *between* t_s and t.
  - *initiates* (3-place, arguments *Event* $\times$ *Fluent* $\times$ *Time*); a grounded instance *initiates*(e, f, t) represents that at any time t, the occurrence of an event e would cause f to begin to hold.
  - *initiates* (3-place, arguments *Occurrence* $\times$ *Fluent* $\times$ *Time*); this is similar to the previous, except that the event e is an action by a subject, regulated by the PDP.
  - *terminates* (3-place, arguments *Event* $\times$ *Fluent* $\times$ *Time*); a grounded instance *terminates*(e, f, t) represents that at any time t, the occurrence of an event e would cause f to cease holding.
  - *terminates* (3-place, arguments *Occurrence* $\times$ *Fluent* $\times$ *Time*); this is similar to the previous, except that the event e is an action by a subject, regulated by the PDP.
- A set $\mathcal{L}^D_{stat}$ of *non-regulatory static predicates*, used to represent unchanging properties of policy-regulated systems. ⌟

Now, we yoke the two kinds of language together, to give the species of first-order sorted language we will use in succeeding sections to reason about the interaction of polcies with domains, and analyse properties of policies.

Definition 3 A *policy representation language* $\mathcal{L} = \mathcal{L}^\pi \cup \mathcal{L}^D$ contains at least the sorts *Time*, *Subject*, *Target*, *Action*, *Event*, *Fluent*, *Action*, where:

- $\mathcal{L}^\pi$ is a policy analysis language, according to Definition 1, with sorts *Subject*, *Target*, *Action*, *Time*.
- $\mathcal{L}^D$ is a domain description language as in Definition 2, with sorts *Fluent*, *Event*, *Action* and *Time*.
- $Occurrence = \{Sub{:}Tar{:}Act \mid Sub \in Subject, Tar \in Target, Act \in Action\}$ ⌟

4 Axioms and Policies

Having given the rudiments of the syntax of our language in Section 3, we now set out the axioms used to represent policies and system behaviour, and provide examples of how our language can be used to analyse policies.

4.1 Authorizations

Before we formally introduce authorization policy rules, we consider the following, as an example of the kind of policy we wish to represent:

Example 4 "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."[2] ⌟

In this natural-language expression of an authorization policy, three actions are mentioned: *notify*, *delete* and *retain*(possibly, with parameters). *delete* should have an argument for a term representing a file, as should *retain*. We will use a fluent $\mathit{fileDesc}(S, F, \mathit{class})$ to represent that file F was supplied by S and has the status of being classified, leaving further details of the arguments to the reader's imagination. *fileDesc* expresses changing properties of the system and so will be wrapped in a *holdsAt* predicate. We will thus represent Example 4 as follows:

$$\begin{aligned}&\mathit{permitted}(\mathit{alice}, \mathit{device}, \mathit{delete}(F), T) \leftarrow \\ &\quad \mathit{holdsAt}(\mathit{fileDesc}(S, F, \mathit{class}), T_n), \\ &\quad \mathit{do}(\mathit{alice}, S, \mathit{notify}(\mathit{delete}(F)), T_n), \\ &\quad T = T_n + 10, \\ &\quad \textbf{not } \mathit{reqInBetween}(S, F, \mathit{retain}(F), T_n, T).\end{aligned}$$

Before we can formally define the shape of authorization policy rules, we must introduce the concept of a *time constraint*.

Definition 5 A *time constraint* C is an expression of the form $\tau_1\ \mathit{op}\ \tau_2$, where each τ_i is a constant or variable of type *Time*, or an arithmetic expression built using $+$, $-$, $/$, $*$, *Time* constants and variables, and where *op* is one of $=, \neq, <, \leqslant, >, \geqslant$. ⌟

Definition 6 A *positive authorization policy rule* is a formula of the form

$$\begin{aligned}&\mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\ &\quad L_1, \ldots, L_m, \\ &\quad C_1, \ldots, C_n.\end{aligned}$$

where

1. the L_i are atoms or atoms preceded by the negation-by-failure **not**, taken from the set
$$\mathcal{L}^{\pi} \cup \mathcal{L}^{D}_{stat} \cup \{\mathit{holdsAt}, \mathit{happens}, \mathit{broken}\};$$
the C_i are time constraints;
2. $m, n \geqslant 0$;
3. any variable appearing in a time constraint must also appear somewhere other than in a time constraint;
4. *Sub*, *Tar*, *Act*, T are terms of type *Subject*, *Target*, *Action* and *Time* respectively;

[2] Such a rule could clearly be expressible in a standard policy language such as XACML [OAS05].

5. 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 a regulatory output predicate; if $L_i \in \mathcal{L}^D_{EC}$, then it should either be *holdsAt* or *broken*.

(Additional constraints of local stratification will be imposed when we gather together policy rules into policies.) ⌟

Condition 1, on the predicates which may appear in the body of our positive authorization policy rules, excludes the *initiates* and *terminates* predicates which are used to express the laws of system evolution—for although policy decisions may depend on what happens in the system, and on which state the system is at a given time, we do not see a place for allowing policies to depend on the laws governing the system. Condition 5, on the time arguments of our predicates, is imposed to ensure that the permission to execute an action in a given state of a trace does not depend on 'future' properties of that trace. It is clear that the example formalized authorization policy rule above fits the conditions of this definition.

Definition 7 A *negative authorization policy rule* is a formula of the form

$$\begin{aligned} &\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\ &\quad L_1, \ldots, L_m, \\ &\quad C_1, \ldots, C_n. \end{aligned}$$

where all the conditions 1–5 hold, as for Definition 6. ⌟

This definition is parallel to that for positive authorization policy rules. An *authorization policy rule* is either a positive or negative authorization policy rule.

When gathering together authorization policy 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 for permission 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. As these behaviours vary between different policy systems, in some being configurable, we make this type of rule optional; some examples of typical rules which may be included are given below.

Another kind of rule is always included, and is used to specify the behaviour of some of the predicates introduced in Section 3. The predicate *reqInBetween*, for instance, is used to encode something related to the operator SINCE of modal temporal logics (see e.g. [Gol92])—an instance $\mathit{reqInBetween}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T', T)$ represents that at time T, a request on the part of *Sub* to perform *Act* on *Tar* was made since time $T' < T$ (so that we use an explicit term of the *Time* sort, rather than a propositional variable, to mark the time since when a request was made; for us, in addition, only a sigle request needs to have been made—this also differs from the case of the modal operator). The utility of having a predicate related to SINCE has been demonstrated to us repeatedly in many formalizations of policy rules, and to capture its semantics, the following rule is always included in our framework.

[3] The symbol $\models$ has the standard meaning of logical consequence.

Definition 8 The *req-in-between* rule is the following formula:

$$\begin{aligned} &\mathit{reqInBetween}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T', T) \leftarrow \\ &\quad \mathit{req}(\mathit{SubTarAct}, T_r), \\ &\quad T' \leqslant T_r \leqslant T. \end{aligned}$$

⌟

Something related to the modal temporal operator sometimes represented as P (where $\mathsf{P}p$ means that p is true at some time previously) may be expressed in relation to *req* by writing $\mathit{reqInBetween}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, 0, T)$—as, for us, time does not extend infinitely far into the past but has a first instant, 0. An instance $\mathit{reqInBetween}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, 0, T)$ can be taken as representing that a request (with the relevant parameters) was made at *some* time before T.

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. Consider the following three examples of additional, optional rules:

Definition 9 The *basic availability rule* is given by:

$$\begin{aligned} &\mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\ &\quad \mathit{req}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T), \\ &\quad \mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T). \end{aligned}$$

The *positive availability rule* is:

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

The *negative availability rule* is:

$$\begin{aligned} &\mathit{deny}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\ &\quad \mathit{req}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T), \\ &\quad \mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T). \end{aligned}$$

⌟

The availability rules given in Definition 9 express different behaviour for the PEP. The first, basic availability, rule, is in a sense more stringent: according to it, an action is enforced by the PEP only when it has been positively permitted by the PDP (i.e. only when an atom $\mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T)$ is true, with appropriate groundings)—this is similar to the default policy used in SELinux [LS01]. The second, alternative 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 final, 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, to record when the result of the PDP is negative, possibly with a view to allowing policies to censure users who make repeatedly denied requests for actions.

Definition 10 A *policy regulation rule* has one of the predicates *do* or *deny* and a body as in Definition 6. ⌟

Many more policy regulation rules are possible than those given as examples in Definition 9; all are optional inclusions in an authorization policy.

Definition 11 An *authorization policy* is a set Π of authorization policy rules, together with the req-in-between rule, and a set of policy regulation rules, such that Π is locally stratified. ⌟

Notice that it is possible to add general authorization policy rules to a policy, which enable a representation of very fine-grained control over defaults for responses to requests. For example, suppose an authorization policy Π contains the positive availability rule, so that actions which are not expressly denied are always executed. There may be a special class of *critial* actions which should be exempted from this liberal policy, and for which we should have express, positive permission. To capture this requirement, we could simply include a rule such as the following:

$$\begin{aligned} &\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\ &\quad \mathit{category}(\mathit{Act}, \mathit{critical}), \\ &\quad \textbf{not } \mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T). \end{aligned}$$

4.2 Separation of Duty and Chinese Wall Policies

It is frequently necessary to write policy rules where the authorization of an action depends on whether or not another, related action is permitted, or has been performed.

Static separation of duty policies typically state that subjects fulfilling a certain role (with which are associated a number of permissions) may not also fulfil another, specified role—and so are not permitted to perform the actions associated with that role. For example, an agent who assigned to the role of 'detonation enabler' in a nuclear arsenal, may be excluded from fulfilling the role associated with actually detonating the bombs. The 'static' qualifier indicates that roles are in this case assigned once and for all, and so the actions which are 'separated' by policies—i.e., which are not jointly permitted—can be determined offline. *Dynamic* separation of duty policies are more flexible; here, where roles (and associated permissions) A and B conflict (or are 'separated'), an agent may be assigned to A in one session, or in relation to one object (and thereby barred from being assigned to B in that session, or on the object), but assigned to B on another object. As the assignment occurs at runtime, the permissions can only be evaluated once the system is up—hence the 'dynamic' qualifier. Finally, *Chinese Wall* policies increase the flexibility further. In a Chinese Wall policy the permission to perform an action depends not on whether, in relation to that action and the object on which it is performed, the subject has been assigned to the relevant role (as in dynamic separation-of-duty policies). Instead, the permission to perform an action is dependent on whether another action has actually been performed.

It is interesting to note that in our system, Chinese Wall policies, which have traditionally been seen as the most flexible of the three sorts of separation-of-duty policies described above, are in many ways the most easy to represent. Suppose, for instance, we have a policy rule such as the following

Example 12 "Whenever a person arms a bomb, they are not permitted to detonate it, and vice versa." ⌟

This can be represented as the following two negative authorization rules:

$$\begin{aligned} &\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{arm}, T) \leftarrow \\ &\quad \mathit{bomb}(\mathit{Tar}), \\ &\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{detonate}, T'), \\ &\quad T' < T. \end{aligned}$$

$$\begin{aligned} &\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{detonate}, T) \leftarrow \\ &\quad \mathit{bomb}(\mathit{Tar}), \\ &\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{arm}, T'), \\ &\quad T' < T. \end{aligned}$$

As can be seen, no extra formal machinery is needed to cope with this paradigmatic case of a Chinese Wall policy, and the formulas above seem an obvious, natural way to formalize the English.

Static and dynamic separation-of-duty policies typically involve the linking of actions with *roles* which a subject may fulfil; the roles are often determined by the associated actions. This may easily be done by introducing terms such as $\mathit{role}(\mathit{Sub}, R)$ to the set of fluents, which represents that the subject Sub fulfils role R. A further fluent $\mathit{role\_action}(R, \mathit{Action})$ can be used to represent which actions are associated with which roles. The value of these fluents can change over time, with relevant actions being introduced into a particular domain in order to represent the actions which assign subjects to, or remove them from, roles, and the same for the assignment of actions to roles.

Once such a machinery of role-assignment, and action-assignment (to roles) is in place, then for *static* separation-of-duty, one can write negative authorization policy rules stating that the assignment of a subject to a role is to be denied, if that subject already fulfils a conflicting role—and similarly for actions. These policy rules express the static separation of duties. In the case of *dynamic* separation-of-duty policies, this simply becomes relativized to the target upon which the action is to be performed (or in some other way, such as being made to depend on the identity of the current session)—such relativization can be effected by the addition of parameters to actions or role names. For the moment, we pass over the details; a related example is presented in Section 5.

In the case where there are two roles whose duties must be separated (either statically or dynamically), we believe the sort of approach we have outlined here works. However, in general there may be $k \geqslant 2$ roles, or subjects, whose duties must be separated [LW06], and in that case a naive extension of our approach will tend to involve writing a number of rules exponential in the size of k. Thus, consider the case where there are 4 roles,

$$\mathit{doctor}, \mathit{nurse}, \mathit{surgeon}, \mathit{admin}$$

whose duties must be separated, so that if an individual is permitted to fulfil the actions of one role, that individual is to be denied requests to fulfil the actions of

another role. This is plausibly expressed by the negative authorization policy rules:

$$
\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{hasrole}(\mathit{Sub}, \mathit{doctor}), T), \\
&\quad \mathit{holdsAt}(\mathit{role\_action}(\mathit{nurse}, \mathit{Act}), T).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{hasrole}(\mathit{Sub}, \mathit{doctor}), T), \\
&\quad \mathit{holdsAt}(\mathit{role\_action}(\mathit{surgeon}, \mathit{Act}), T).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{hasrole}(\mathit{Sub}, \mathit{doctor}), T), \\
&\quad \mathit{holdsAt}(\mathit{role\_action}(\mathit{admin}, \mathit{Act}), T).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{hasrole}(\mathit{Sub}, \mathit{nurse}), T), \\
&\quad \mathit{holdsAt}(\mathit{role\_action}(\mathit{doctor}, \mathit{Act}), T). \\
&\qquad \vdots
\end{aligned}
$$

And so on—with, in general, $k(k+1)$ rules having to be included.

Another way to approach the case of k different users combined in a separation-of-duty policy is to introduce symbols which can be used to represent lists, and a predicate symbol used to denote list membership. If we imagine a term [] for the empty list, and consider $[\cdot|\cdot]$ as a function symbol which represents the operation of making a list from an element (the *head*) and another list, together with a new predicate *member*, we can write our separation-of-duty policy in one rule as:

$$
\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{hasrole}(\mathit{Sub}, R_1), T), \\
&\quad \mathit{member}(R_1, [\mathit{doctor}, \mathit{nurse}, \mathit{admin}, \mathit{surgeon}]), \\
&\quad \mathit{member}(R_2, [\mathit{doctor}, \mathit{nurse}, \mathit{admin}, \mathit{surgeon}]), \\
&\quad R_1 \neq R_2, \\
&\quad \mathit{holdsAt}(\mathit{role\_action}(R_2, \mathit{Act}), T).
\end{aligned}
$$

(We have used some syntactic sugar in writing the lists.) The separation constraints can now be expressed concisely; the problem is that the introduction of the function symbol appears to take us outside the realm of Datalog, and would increase the computational complexity of policy evaluation when policies are expressed in our formal framework.

4.3 Obligations

There are many different classes of obligations—on properties which should hold at states, or on an agent to perform an action, or an action to be performed regardless of who performs it, and so on. In this paper we follow [IYW06] in limiting our

attention to obligations acquired by a subject to perform an action on a target. The subject could be an entity external to the system—consider the case where a user is allowed to execute an action as long as he or she accepts the obligation to execute another action later[4]—or obligations may also be imposed in parts of the system, the system itself then taking the responsibility for executing the action. In the former case, the system itself cannot enforce the obligation, merely being able to monitor whether the obligation has been fulfilled or not.

Example 13 "A connecting node must provide a second identification within five minutes of establishing a connection to the wireless 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:

$$\begin{aligned} &\mathit{obl}(U, \mathit{server}, \mathit{submit2ID}(U, \mathit{server}), T{+}0.1, T{+}5\mathit{mins}, T{+}0.1) \leftarrow \\ &\quad \mathit{holdsAt}(\mathit{node}(U), T), \\ &\quad \mathit{do}(U, \mathit{server}, \mathit{connect}(U, \mathit{server}), T). \end{aligned}$$

$$\begin{aligned} &\mathit{obl}(\mathit{server}, \mathit{server}, \mathit{disconnect}(U, \mathit{server}), T_e, T_e{+}1, T_e) \leftarrow \\ &\quad \mathit{violated}(U, \mathit{server}, \mathit{submit2ID}(U, \mathit{server}), T_s, T_e, T_e). \end{aligned}$$

As previously, the EC predicate *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.

Definition 14 An *obligation policy rule* is a formula of the form

$$\begin{aligned} &\mathit{obl}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T) \leftarrow \\ &\quad L_1, \ldots, L_m, \\ &\quad C_1, \ldots, C_n. \end{aligned}$$

where all the following hold:

1. the L_i are atoms or atoms preceded by the negation-by-failure **not**, taken from the set
$$\mathcal{L}^{\pi} \cup \mathcal{L}^{D}_{stat} \cup \{\mathit{holdsAt}, \mathit{happens}, \mathit{broken}\};$$
the C_i are time constraints;
2. $m, n \geqslant 0$;
3. any variable appearing in a time constraint must also appear somewhere other than in a time constraint;
4. *Sub*, *Tar*, *Act*, are terms of type *Subject*, *Target* and *Action* respectively; T_s, T_e and T are all of sort *Time*;

[4]This is a generalization of the model of obligations proposed in XACML [OAS05].

5. for the time argument T_i of each $L_i \notin \mathcal{L}^D_{stat}$, we must have $C_1 \wedge \cdots \wedge C_n \models T_i \leqslant T$; if $C_1 \wedge \cdots \wedge C_n \models T_i = T$ then the L_i must not be a regulatory output predicate; if $L_i \in \mathcal{L}^D_{EC}$, then it should either be *holdsAt* or *broken*.

(Note that we do not insist that $t_s < t_e$, but that when $t_s = t_e$, the obligation cannot be fulfilled or violated, and thus *sensible* obligation policy rules will always include constraints which make $t_s < t_e$.) ⌟

The conditions 1–5 here are essentially the same as those given for the definition (6) of a positive authorization policy rule. We now need a number of further rules for specifying the conditions under which an obligation is fulfilled or violated, and defining the subsidiary predicate *cease_obl*.

Definition 15 The predicate *fulfilled* is defined by the following *fulfilment rule*:

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

The predicate *violated* is defined by the *violation rule*:

$$\begin{aligned} &\mathit{violated}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T) \leftarrow \\ &\quad \mathit{obl}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T_{init}), \\ &\quad \textbf{not } \mathit{cease\_obl}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_{init}, T_s, T_e, T_e), \\ &\quad T_{init} \leqslant T_s < T_e \leqslant T. \end{aligned}$$

Two *cease obl rules* are used to define the predicate *cease_obl*:

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

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

It is easiest to discuss *cease_obl* first. This is a state regulatory predicate which is used to mark the fact that something has occurred which would cause an obligation to cease; as there are two occurrences which would make an obligation cease (either its fulfilment, or a revocation of the original obligation), there are two clauses defining *cease_obl*. Fulfilment is the more straightforward: an obligation is fulfilled when the action a subject has been obliged to perform is executed (notice that the *do* in the body of the rule here means that the execution of such an action must first be authorized by the system). The *cease_obl* rule for revocation makes use of the *revoke* members of the sort *Action*, which were introduced in Section 3; revocation occurs when the PDP has authorized the request for a revocation action. Clearly,

this subject might be the one bound by the obligation, a central administrator in the system, or an entirely different agent. The parameters of the *revoke* argument identify the obligation which is to be revoked.

The rules for when an obligation is fulfilled or violated are then defined using *cease_obl*: we need to be able to reason about times after an obligation has been initially contracted, but when it has not yet ceased—this is the source of the negated *cease_obl* predicate in the bodies of the fulfilment and violation rules. An existing obligation to perform *Act* between T_s and T_e is fulfilled when a *do* predicate for that *Act* is true between T_s and T_e; and obligations to perform actions between T_s and T_e are violated, when the time reaches T_e and the action has neither been executed, or the obligation revoked.

Definition 16 An *obligation policy* Π is a set of obligation policy rules, together with the 'fulfilment', 'violation' and 'cease obl' rules, such that Π is locally stratified.⌟

Finally, we collect authorization and obligation policies together, as follows.

Definition 17 A *mixed policy* $\Pi = \Pi_a \cup \Pi_o$ is any union of an authorization policy Π_a and an obligation policy Π_o. (We will frequently drop the qualified 'mixed'.) ⌟

4.4 Event Calculus and Domain Models

We use the *Event Calculus* (EC) to represent and reason about changing properties of the domains regulated by policies which we model. The EC is a well-studied formalism, variants of which exist both as logic programs and in first-order logical axioms (with the use of a second-order axiom enforcing 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. (For the original formulation of the EC as a logic program, see [KS86]; for a comparison of some recent approaches, see [MS02].)

Definition 18 The set *EC* of *Event Calculus core axioms* are as follows:

$$
\begin{aligned}
&\mathit{holdsAt}(F, T) \leftarrow \\
&\quad \mathit{initially}(F), \\
&\quad \textbf{not } \mathit{broken}(F, 0, T).
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
&\mathit{holdsAt}(F, T) \leftarrow \\
&\quad \mathit{initiates}(\mathit{Sub}{:}\mathit{Tar}{:}\mathit{Act}, F, T_s), \\
&\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s), \\
&\quad T_s < T, \\
&\quad \textbf{not } \mathit{broken}(F, T_s, T).
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
&\mathit{holdsAt}(F, T) \leftarrow \\
&\quad \mathit{initiates}(\mathit{Event}, F, T_s), \\
&\quad \mathit{happens}(\mathit{Event}, T_s), \\
&\quad T_s < T, \\
&\quad \textbf{not } \mathit{broken}(F, T_s, T).
\end{aligned} \tag{3}
$$

$$
\begin{aligned}
&\mathit{broken}(F, T_s, T) \leftarrow \\
&\quad \mathit{terminates}(\mathit{Sub}{:}\mathit{Tar}{:}\mathit{Act}, F, T'), \\
&\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T'), \\
&\quad T_s < T' < T.
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
&\mathit{broken}(F, T_s, T) \leftarrow \\
&\quad \mathit{terminates}(\mathit{Event}, F, T'), \\
&\quad \mathit{happens}(\mathit{Event}, T'), \\
&\quad T_s < T' < T.
\end{aligned} \tag{5}
$$

(These are the domain-independent axioms which are included in all formalized systems modelled by the event calculus.) ⌟

The first clause specifies that a changeable property of the system holds at time T, if that property held at time 0, or in the initial state, and nothing disturbed its persistence-by-default. The next two axioms represent how a fluent representing a changeable property comes to be true in a system: by being initiated, either as a consequence of an action enforced by the PDP/PEP, or else 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 one clause for disturbance caused by enforced, regulated actions, and one clause for disturbance caused by unregulated events.

When modelling the systems regulated by a PDP/PEP, we will write down a combination of clauses defining the predicates *initiates* and *terminates*, which describe how actions or events occurring in a system change that systems properties. For example, consider a system where the existence of a file in a directory is represented by a fluent $\mathit{isIn}(F, \mathit{Dir})$, where F is a variable for a filename, and *Dir* for a

directory name. A user S (the *subject*) may make a request to move (the *action*) the file to another directory *Dir* (the *target*); an authorization policy would regulate the permissions for this action, but the following *initiates* and *terminates* actions would be needed to model the effect of a permitted and enforced action on the system:

$$initiates(S{:}Dir{:}move(F), isIn(F, Dir), T).$$

$$terminates(S{:}Dir{:}move(F), isIn(F, Dir'), T) \leftarrow$$
$$holdsAt(isIn(F, Dir'), T).$$

The first rule, together with the EC axiom 2, means that whenever a $do(S, Dir, F, T)$ action is executed by the PEP (following a permitted request to move a file to *Dir*), then this will force $holdsAt(isin(F, Dir), T')$ to be true, for $T' > T$—until the file is moved again. Since the file has been moved, we must also ensure that after a *move* action has been executed, the atom $holdsAt(isin(F, Dir'), T_b)$ is no longer true, where $T_b < T$ and Dir' is the previous locaction of the file: this behaviour is modelled by the *terminates* axiom.

In general, *initiates* and *terminates* axioms (such as those above) are used to model the dynamic properties of a system. *initiates* axioms describe how fluents become true of a system, usually after the execution of an action; and *terminates* axioms specify how fluents become no longer true of a system. In addition, we also use predicates from $\mathcal{L}^D_{stat}$ to represent unchanging properties of systems. 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 19 A *domain description* is a set $D = EC \cup D'$ containing the event calculus core axioms, together with a set D' of formulas of either of the three forms: a *static domain axiom*

$$A \leftarrow L_1, \ldots, L_n.$$

such that L is an atom, and the $L_1, \ldots, L_n$ are literals, of predicates in $\mathcal{L}^D_{stat}$; or *either* an *initiates* axiom

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

or a *terminates axiom*

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

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. ⌟

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

Definition 20 A *domain-constrained policy* $P = \Pi \cup D$ is the union of a mixed policy Π and a domain description D. ⌟

Domain-constrained policies are sets of policy rules (of authorization and obligation policies) together with a specification of the system which these policies regulate. These are the basic sets of formulas upon which our analysis algorithms (described in Section 6) will operate. (Note that if Π and D are stratified, then so is $\Pi \cup D$.)

4.5 Semantics

We use the *stable model* semantics of Gelfond and Lifschitz [GL88], in order to provide a semantics for our domain-constrained policies.

The stable model semantics starts with a (ground) normal logic program P, and a set of ground atoms I. The *reduct* P^I is defined as the result of removing from P any clause which has a negative literal **not** a in its body such that $a \in I$, and deleting from the bodies of the rules remaining in P all other negative literals. That which remains, P^I, is a definite logic program, with no negation-by-failure, and thus has a unique minimal model $min(P^I)$ according to the standard semantics for negation-free logic programs. I is defined to be a stable model of P if $I = min(P^I)$, the term 'stable' indicating the fixpoint definition. The stable model semantics is an acknowledged standard semantics for normal logic programsl; it is know that locally-stratified normal logic programs have a unique stable model.

To capture the operational model described in Section 2, 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. Other predicates are not included here, as the truth or falsity of their ground instances will be determined by the semantics, using the policy and dynamic evolution laws of the 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 21 Let P be a ground, domain-constrained policy (see Definition 20). Then, a *policy-regulated trace* is the stable model of $P \cup \Delta^D \cup \Delta^\pi$.
We let $model(P \cup \Delta^D \cup \Delta^\pi)$ refer to the (unique) stable model of $P \cup \Delta^D \cup \Delta^\pi$. ⌟

5 Examples

Our examples are related to a scenario involving a multi-company team of business people conducting a virtual meeting to collaborate on a project. The project is led by Alice who works for Acme Inc. Other participants are Bob from Boolean plc and Charles from ComputaTech who is working from home. In addition to supporting voice and text communications, the virtual meeting software being used allows meeting participants to share documents with each other. Documents can be classified as 'Public' (can be shared with anyone), 'Project' (can be shared with project members) or 'Company' (can only be shared within the company). The software is integrated with a policy-based security management framework that ensures that the security policies of the individuals and companies are enforced. Based on this scenario, we have selected a few policies that satisfy a range of security requirements, from simple access control to separation of duty. For each policy, we present a natural language and formal definition together with a discussion of the formal language features that are being used:

Example 22 "Project partners who are permitted to read the meeting agenda are allowed to read related project documentation 24 hrs before the meeting starts." ⌟

This policy is an authorization rule that depends on another permission. We would express this rule in our formalism as follows:

$$
\begin{aligned}
&permitted(Subject, D, read, T_2) \leftarrow \\
&\quad project(P), \\
&\quad meeting(M, P), \\
&\quad agenda(M, A), \\
&\quad permitted(Subject, A, read, T_1), \\
&\quad projectDoc(D, P), \\
&\quad startTime(M, T_3, T_2), \\
&\quad T_1 < T_2 < T_3, \\
&\quad (T_3 - T_2) < 24hrs.
\end{aligned}
$$

Despite having *permitted* in both the head and body of the rule, the above policy is acceptable in our formalism because the permitted predicate in the body specifies a target that is disjoint from the target used in the head predicate. (We have simplified matters somewhat in presuming many of these predicates to be static.)

The following rule is an instance of dynamic separation-of-duty.

Example 23 "Standard meeting attenders are allowed to vote at meetings; meeting administrators are allowed to view those votes. The meeting chair is allowed to allocate members of the company to these roles. Nobody is permitted to be both a standard meeting attender and a meeting admin." ⌟

This requires a more complex formalization:

$$\begin{aligned}
&\mathit{permitted}(\mathit{Sub}, M, \mathit{vote}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(S, \mathit{standard_attender}(M)), T).
\end{aligned}$$

$$\begin{aligned}
&\mathit{permitted}(\mathit{Sub}, M, \mathit{view_votes}, T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(S, \mathit{meeting_admin}(M)), T).
\end{aligned}$$

$$\begin{aligned}
&\mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{allocate}(\mathit{standard_attender}(M)), T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(S, \mathit{chair}(M)), T).
\end{aligned}$$

$$\begin{aligned}
&\mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{allocate}(\mathit{meeting_admin}(M)), T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(S, \mathit{chair}(M)), T).
\end{aligned}$$

$$\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{allocate}(\mathit{standard_attender}(M)), T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(\mathit{Tar}, \mathit{meeting_admin}(M)), T).
\end{aligned}$$

$$\begin{aligned}
&\mathit{denied}(\mathit{Sub}, \mathit{Tar}, \mathit{allocate}(\mathit{meeting_admin}(M)), T) \leftarrow \\
&\quad \mathit{holdsAt}(\mathit{role}(\mathit{Tar}, \mathit{standard_attender}(M)), T).
\end{aligned}$$

$$\mathit{initiates}(\mathit{Sub}{:}\mathit{Tar}{:}\mathit{allocate}(R), \mathit{role}(\mathit{Tar}, R), T).$$

The first two positive authorization policies specify the permissions of those fulfilling the roles of *standard_attender* and *meeting_admin*. The next two rules state the permissions of the chair of a meeting in allocating roles. Then, the two negative authorization policy rules express the separation of duties: if a target is a meeting administrator, they are not to be permitted to become a standard attender, and vice versa. Finally, the effects of an allocation-to-role action are given, in the *initiates* axiom.

Example 24 "A meeting attendee is not allowed to classify a document as 'Public' if they created the document." ⌟

This policy ensures that the decision to declassify a document is not taken by the person who created the document. It is an example of a separation-of-duty policy where a subject is prevented from performing a particular action if he has performed some other conflicting action previously. The separation can be expressed as follows:

$$\begin{aligned}
&\mathit{denied}(\mathit{Sub}, D, \mathit{classify}(\mathit{public}), T_2) \leftarrow \\
&\quad \mathit{project}(P), \\
&\quad \mathit{meeting}(M, P), \\
&\quad \mathit{attendee}(\mathit{Sub}, M), \\
&\quad \mathit{doc}(D, M), \\
&\quad \mathit{do}(\mathit{Sub}, D, \mathit{create}, T_1), \\
&\quad T_1 < T_2.
\end{aligned}$$

Example 25 "Alice is allowed to create and declassify project documents, but not on the same project." ⌟

We read this is an example of a Chinese Wall policy where a subject is prohibited from performing an action on a particular target if he has already performed some other conflicting action on the same, or another but related target (in this case, another document on the same project). We formalize as follows:

$$
\begin{aligned}
&\mathit{permitted}(\mathit{alice}, D, \mathit{create}, T) \leftarrow \\
&\qquad \mathit{project}(P), \\
&\qquad \textbf{not } \mathit{denied}(\mathit{alice}, D, \mathit{create}, T).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{permitted}(\mathit{alice}, D, \mathit{classify}(\mathit{public}), T) \leftarrow \\
&\qquad \mathit{project}(P), \\
&\qquad \mathit{project\_doc}(P, D), \\
&\qquad \textbf{not } \mathit{denied}(\mathit{alice}, D, \mathit{classify}(\mathit{public}), T).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denied}(\mathit{alice}, D_1, \mathit{create}, T_2) \leftarrow \\
&\qquad \mathit{project}(P), \\
&\qquad \mathit{project\_doc}(P, D_1), \\
&\qquad \mathit{project\_doc}(P, D_2), \\
&\qquad T_1 < T_2, \\
&\qquad \mathit{do}(\mathit{alice}, D_2, \mathit{classify}(\mathit{public}), T_1).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denied}(\mathit{alice}, D_1, \mathit{classify}(\mathit{public}), T_2) \leftarrow \\
&\qquad \mathit{project}(P), \\
&\qquad \mathit{project\_doc}(P, D_1), \\
&\qquad \mathit{project\_doc}(P, D_2), \\
&\qquad T_1 < T_2, \\
&\qquad \mathit{do}(\mathit{alice}, D_2, \mathit{create}, T_1).
\end{aligned}
$$

In these policy rules, the permissions are expressed and given conditions (the literals **not** *denied* in their bodies) which express the exceptions to them. (There are a number of different ways to represent this exception structure; we have chosen a concise one for the purposes of illustration.) This is close to how we intuit the meaning of the English expression of the policy. The circumstances under which the exceptions arise—which are the cases where there is a conflict in the separation-of-duty policy rules—are then specified by the negative authorization rules.

6 Policy Analysis

Using our formalism, policies are always classically consistent—as the models for our policies (the policy-regulated traces specified in Definition 21) are stable models, and hence sets of atoms, we cannot, somehow, have both $\mathit{permitted}(\mathit{sub}, \mathit{tar}, \mathit{act}, t)$ and **not** $\mathit{permitted}(\mathit{sub}, \mathit{tar}, \mathit{act}, t)$ in a policy-regulated trace σ, for given subject

sub, target *tar*, action *act* and time t; the same, of course, applies to obligation policies. However, other forms of analysis and conflict detection, not focused on classical inconsistency, are relevant to our formalization; in this section we describe the kinds of properties—or, more significantly, their absence—we look for, together with the analysis algorithms used.

Thus, *Modality Conflicts* may be of several different kinds. One form arises when, for the same terms *sub*, *tar*, *act* and t, the atoms $\mathit{permitted}(\mathit{sub}, \mathit{tar}, \mathit{act}, t)$ and $\mathit{denied}(\mathit{sub}, \mathit{tar}, \mathit{act}, t)$ are both contained in a policy-regulated trace for some domain-constrained policy P; the significance of this is that the authorization-policy component of the policy P both permits and denies a request from the subject to perform the action, the permission and denial occurring at the *same* time. Another kind of modality conflict arises when there is an obligation upon a subject to perform an action between two times, but at no time between does the subject have permission. Here, given a domain-constrained policy P, there would be sets Δ^D and Δ^π, such that for the policy-regulated trace M (the stable model of $P \cup \Delta^D \cup \Delta^\pi$), we have:

- $\mathit{obl}(\mathit{sub}, \mathit{tar}, \mathit{act}, t_s, t_e, t_{init}) \in M$;
- $\mathit{cease\_obl}(\mathit{sub}, \mathit{tar}, \mathit{act}, t_{init}, t_s, t_e, t) \notin M$, for all t such that $t_{init} \leqslant t \leqslant t_e$;
- $\mathit{denied}(\mathit{sub}, \mathit{tar}, \mathit{act}, t) \in M$, for all t such that $t_s \leqslant t < t_e$.

Modality conflicts such as these may arise for different sorts of reason. Clearly, if an authorization policy Π contains, for example, the two policy rules

$$\mathit{permitted}(\mathit{client}, \mathit{server}, \mathit{connect}, T) \quad \text{and} \quad \mathit{denied}(\mathit{client}, \mathit{server}, \mathit{connect}, T),$$

then, no matter what the domain description to which the policy system is yoked, and no matter what the series of input requests or non-regulated actions of the system, any policy-regulated trace being a model of $\Pi \cup \Delta^D \cup \Delta^\pi$ will have a modality conflict; here we might speak of the modality conflict's being *intrinsic* to the policy rules. On the other hand, consider the case of an authorization policy containing the rules:

$$\begin{aligned} &\mathit{permitted}(\mathit{doctor}, \mathit{patient}, \mathit{cut}, T) \leftarrow \\ &\qquad \mathit{holdsAt}(\mathit{anaesthetized}(\mathit{patient}), T). \end{aligned}$$

$$\begin{aligned} &\mathit{denied}(\mathit{doctor}, \mathit{patient}, \mathit{cut}, T) \leftarrow \\ &\qquad \textbf{not } \mathit{holdsAt}(\mathit{seriousCondition}(\mathit{patient}), T). \end{aligned}$$

In this case, the possibility of a modality conflict will depend on what the domain description for the governed system is. If the domain-constrained policy contains the further rules

$$\mathit{initiates}(\mathit{anaesthetist}{:}P{:}\mathit{drug}, \mathit{anaesthetized}(P), T).$$

$$\begin{aligned} &\mathit{permitted}(\mathit{anaesthetist}, P, \mathit{drug}, T) \leftarrow \\ &\qquad \mathit{holdsAt}(\mathit{seriousCondition}(P), T). \end{aligned}$$

then (supposing certain further properties) it might easily be true that a situation may never arise in which a patient whose condition is not serious is anaesthetized—in

this case, the modality conflict suggested as possible by the initial two policy rules may be shown not to occur, under any circumstances. The analysis tools we use take into account the content of the domain description to give an accurate analysis of the circumstances (if any) under which a modality conflict can arise, giving the relevant information about those circumstances to the user, and also the policy rules used to derive the conflict.

Several other types of analysis are possible. Thus, *Coverage Analysis* concerns whether the policy rules in a given policy, in combination with the domain description, determine responses from the PDP/PEP in all cases of interest to requests for permission to perform an action. One may be interested in proving that in all systems of a given configuration, a user *Alice* has the right to perform a given action—it is easy to test for such a property using our formalism and algorithms. We may also wish to test for other properties, such as whether a given set of permissions and obligations are compossible, or whether a certain system state is reachable—the output from the algorithm we use would be, not merely an answer in the affirmative, but a trace of requests made and non-regulatory events occuring which leads to the combination of policy decisions or system properties.

6.1 Abductive Constraint Logic Programming

The task of analysing a domain-constrained policy $P = \Pi \cup D$ to see, for instance, whether there are no modality conflicts, 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 that

$$\forall T(\textbf{not } (\mathit{permitted}(\mathit{sub}, \mathit{tar}, \mathit{act}, T) \wedge \mathit{denied}(\mathit{sub}, \mathit{tar}, \mathit{act}, T))) \tag{6}$$

for given ground terms *sub*, *tar*, *act*. This expression states that a certain sort of modality conflict between authorization and denial never occurs. 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; a system designer can then decide whether or not the particular series of inputs which give rise to the modality conflict may be ignored, or alternately, can modify the policy in the light of the conflict found, and the specific series of inputs which give rise to it.

Checking whether the system verifies the property formalized above, therefore converts into the task of checking whether there are inputs Δ^D and Δ^π (as described in Section 4.5) such that the property is *not* true, i.e. whether there are sets such that

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

This problem is equivalent to showing that the formula above (6) is false, and can be solved using *Abductive Logic Programming* (with constraints), the algorithm for which attempts to compute the sets Δ^D and Δ^π. 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 framework based on that found in [KMM00], details of which are now presented.

First, the set of *abducible predicates* is defined to consist of

$$\mathit{initially}, \mathit{happens}, \mathit{req},$$

together with the predicates from $\mathcal{L}^D_{stat}$ *not* appearing in the heads of rules. All other predicates from the language $\mathcal{L} = \mathcal{L}^\pi \cup \mathcal{L}^D$ are *defined*. An analysis task is defined as a tuple (P, A, G), where P is a domain-constrained policy, A is the set of abducible predicates, and G is the set $\{C, L_1, \ldots, L_n\}$ of literals $L_1, \ldots, L_n$ appearing in the property we want to try and prove,[5] together with a (possibly empty) set of time-constraints on variables appearing in the literals $L_1, \ldots, L_n$. Given an analysis task, the abductive proof procedure takes as input the rules P, the time constraints C (if any), the literals $L_1, \ldots, L_n$ and a set Δ of abducibles. The literals L_i are either ground or Skolemized on their time variables, and the set Δ is, at the beginning of the proof procedure, empty. The abductive proof generates as output a set C' of time constraints (which imply C) and a set Δ' of abducibles which, instantiated on any solution of C', gives a set Δ^* of ground abducibles such that

$$model(P \cup \Delta^*) \models (C \wedge L_1 \wedge \cdots \wedge L_n)\sigma,$$

where σ is a substitution of the free variables occurring in $\{C, L_1, \ldots, L_n\}$. The set Δ is called an *abductive explanation* for $C \wedge L_1 \wedge \cdots \wedge L_n$.

The abductive proof procedure is composed of two modules, the *abduction phase* and *consistency phase*, that interleave with each other. The former is based on SLD resolution, a standard proof procedure for logic programs. It 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 SLD 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 set of hypothesis Δ. The consistency check is important not only for the consistency of Δ but also for the consistency of $\Delta \cup P$. During abduction, negated non-abducible predicates are also added to Δ (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 one separate branch of computation for each resolvent 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 are made to fail) the calling abductive computation continues with the abducible A added to Δ (along with any other abducibles accumulated during the consistency computation) and the constraint store C (along with any other time constraints accumulated during the consistency computation). If some branch of the consistency computation does not succeed (i.e. it cannot be made to fail) the calling abductive computation fails, indicating that A is inconsistent with $\Delta \cup P$. In order to ensure consistency across its different branches of computation, the consistency module keeps track of constraints (IC^*) related to abducible predicates that unify with elements in Δ. These can be seen as universally quantified assumptions about the abducibles in Δ, dynamically generated during local consistency computations and which must hold for Δ to be an abductive explanation consistent with P. As the consistency computation can interleave with the abductive computation, the

[5] In the example treated earlier, for instance, G would be the set containing the two atoms $permitted(sub, tar, act, T)$ and $denied(sub, tar, act, T)$.

set IC^* of dynamically generated constraints is also assumed to be a parameter of the abduction module. The abduction and consistency modules are described in Figures 2 and 3.[6]

Abduction$(G, C, \Delta, IC^*, \Pi)$:
{*returns a new* Δ, *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 Π where H, L_i unify with unifier θ, 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 Δ with unifier θ, then
   Let $G = (G \setminus \{L_i\})\theta$;
4. If L_i is a literal with an abducible predicate that does not unify with any element in Δ then
   Skolemize L_i into S_i, and constraint $C^{'}$
   if $S_i^* \in \Delta$ then **return failure**
   else
   Let $\Delta = \Delta \cup \{S_i\}$; $G = G \setminus \{L_i\}$; $C = C \cup C^{'}$;
   Let $F =$ **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 Δ, C and IC^*
end Abduction

Figure 2: Abduction Procedure

The abduction module 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 Δ of abducibles, a set IC^* of dynamically-generated constraints, and P, the domain-constrained policy. At the beginning both Δ and IC^* are empty. The module can either fail or terminate successfully, in which case it returns an updated Δ, C and IC^*.

The consistency module takes as input a set $\{F_1, \ldots, F_n\}$ of the literals to be

[6] **Reduction**$(IC^*, L) = \{(\{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n\}, C)\theta \mid (\{L_1, \ldots, L_n\}, C) \in IC^*$ and $L_i\theta = L\theta\}$

Consistency$(F, C, \Delta, IC^*, \Pi)$:
{*returns a new* Δ, *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 $\phi \wedge C^{'} \rightarrow H \in \Pi$ such that H and L_i unifies with unifier θ do
   if ϕ 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 θ 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 Δ, then
   if **Abduction**$(\{L_i^*\}, C, \Delta, IC^*, \Pi)$ returns $\Delta^{'}$, $C^{''}$
   and IC_1^* then
   Let $\Delta = \Delta^{'}$, $C = C^{''}$ and $IC^* = IC_1^*$;
   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 Δ, C and IC^*
end Consistency

Figure 3: Consitency Procedure

checked for consistency, a (possibly empty) set C_{loc} of time constraints, a set Δ of abducibles, and a set IC^* of dynamic constraints and P. The module can either fail or terminate successfully, in which case it outputs $\Delta^{'} \supseteq \Delta$, $C^{'} \supseteq C$ and IC^*.

As an example, consider a *coverage* analysis of the policy Example 4. For ease of readability, we simplify the notation of the authorization rule in formalizing this

policy to the following:

$$\begin{aligned} &\mathit{reqInBetween}(X, T_0, T_2) \leftarrow \\ &\quad \mathit{req}(X, T1), \\ &\quad T_0 \leqslant T1, \\ &\quad T_1 \leqslant T2. \end{aligned} \tag{7}$$

$$\begin{aligned} &\mathit{permitted}(\mathit{delete}, T_1) \leftarrow \\ &\quad \mathit{do}(\mathit{notify}, T_0), \\ &\quad T_1 = T_0 + 10\mathit{mins}, \\ &\quad \textbf{not } \mathit{reqInBetween}(\mathit{retain}, T_0, T_1). \end{aligned} \tag{8}$$

$$\begin{aligned} &\mathit{do}(X, T) \leftarrow \\ &\quad \mathit{req}(X, T), \\ &\quad \mathit{permitted}(X, T). \end{aligned} \tag{9}$$

$$\mathit{permitted}(\mathit{notify}, T). \tag{10}$$

(Thus, we imagine that the policy rules are written without reference to predicates defined by the domain description.) We can check that the action *delete* is permitted at some time point T_f; this amounts to performing an abductive task with $G = \{\mathit{permitted}(\mathit{delete}, T_f)\}$, $C = \{\}$, $\Delta = \{\}$ and $IC = \{\}$. Without going through the details of the way the algorithm applies itself, we can note that, intuitively, $\mathit{permitted}(\mathit{delete}, T_f)$ is first attempted to be proved using the Abduction module; a resolution step with the head of the rule adds the constraint $T_f = T_f + 10$ to the store (after relevant variable bindings) and the remaining literals in the body of rule 8 to the list of literals to be proved. In order to show that $\mathit{do}(\mathit{notify}, T_0)$, we abduce $\mathit{req}(\mathit{notify}, t_0)$, where t_0 is a Skolemized variable for T_0. We must then consider the remaining literal in the body of rule 8, which is **not** $\mathit{reqInBetween}(\mathit{retain}, t_0, T_f)$. As this is a non-abducible negative literal, it is covered by case (5) of the Abduction module, which runs a Consistency module check on the Skolemized complement $\mathit{reqInBetween}(\mathit{retain}, t_0, t_f)$. The consistency check identifies $\mathit{reqInBetween}(\mathit{retain}, t_0, t_f)$ as a non-abducible, and resolves this with the head of rule 7, leaving the consistency derivation to continue with the body of that rule. Now, either we try to fail by choosing the literal $\mathit{req}(\mathit{retain}, t_0, T_3)$ (after renaming), which will close the consistency phase as there is no literal in the current Δ to unify with; or else, we choose one of the constraints, and find a C' in accordance with case (7). In either case, the consistency phase terminates, and thus the outer abduction module is also done. The solution found contains the literals

$$\mathit{req}(\mathit{notify}, t_f) \qquad \text{and} \qquad \textbf{not } \mathit{reqInBetween}(\mathit{retain}, t_0, t_f),$$

where t_0 and t_f are Skolemized versions of the variables T_0 and T_f, and also the constraint

$$t_f = t_0 + 10.$$

As has been described above, analyzing a set S of authorization policies for *one* kind of *modality* conflict is defined as showing that the following property holds for

some subject *Sub*, target *Tar* and action *Act*, and for all sets of the relevant kind Δ^D and Δ^π:

$$model(P \cup \Delta^D \cup \Delta^\pi) \models \forall T{:}Time \ \textbf{not} \ (permitted(Sub, Tar, Act, T) \\ \wedge \ denied(Sub, Tar, Act, T))$$

The abductive proof procedure procedure checks whether it is possible to identify a (set of) input formulae $\Delta = \Delta^D \cup \Delta^\pi$ of the form (**not**) $req(S, Ta, A, T)$, and domain dependent facts (if needed), such that

$$model(P \cup \Delta^D \cup \Delta^\pi) \models permitted(Sub, Tar, Act, T) \wedge denied(Sub, Tar, Act, T)$$

If such a computation, which in this case always terminates, fails then it can be assumed that the policy, together with the given system description, has no modality conflict of the relevant kind. On the other hand, if the abductive proof procedure computes such a Δ, P is said to *imply a modality conflict in the system* D.

The same technique can be applied for the other type of modality conflict—that involving an obligation in the absence of related permissions. Indeed, an outstanding benefit of using the form of abductive constraint logic programming we employ is that it is highly general with respect to the class of properties for which we can analyze our policies and systems.

6.2 Termination and Complexity

We can 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 algorithms just presented.

We insist that the language we use (the sorts *Subject*, *Target*, *Action*, *Fluent*, *Event*) is finite.

If we stipulate that the models of a domain-constrained policy $P = \Pi \cup D$ must be such that in the mixed 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 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

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

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 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 dependencies amongst the members of our mixed policies (see Definition 17). 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, 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.)

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.

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) independent of the analysis. Current implementations of abduction are more general than required by our analyses.

6.3 Implementation

We have built a system which implements our formal framework for policy analysis, enabling us to test the correctness of many of the ideas which underlie the choice of language details, and the form which axioms take. It is freely available to download from `http://www.doc.ic.ac.uk/~rac101/ffpa/`.

The implementation uses the ASystem [VN04], a free and open-source program written in Prolog, for the abductive constraint logic-programming component of the analysis algorithms. The tests have enabled us to find modality conflicts, coverage problems, and another of other interesting properties of policies in conjunction with system descriptions.

One difference between this implementation and the details presented in the current paper is that the constraint solver used by the ASystem is based on finite domains. For this reason, we adapted our axioms to work on an integer base for the *Time* sort, and chose a maximum time to consider in order to make the *Time* domain finite. In all cases we have examined, analysis results achieved under these modifications would hold under the original version of the axioms with $\mathbb{R}$ as the time sort.

It is not essential to use a finite-domain solver for the constraint-satisfaction part of our abductive algorithm; the abductive logic-programming framework we have presented in this section 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.

Sample code for an ASystem implementation of our framework, together with a test domain and policies, are presented in the Appendices.

7 Related Work

The work presented in this paper bears some similarities to previous work on formalizing policies using the Event Calculus [Ban05]. Both approaches advocate modeling the temporal properties of the managed system and they both support authorization and obligation policy rules. However, the formalization in [Ban05] implicitly focuses on modeling the Ponder language.[7] This results in many restrictions, amongst which the following three are most notable: fixing an open policy for authorizations (i.e. actions not explicitly denied are permitted), not having conflict resolution mechanisms by assuming that all conflicts are resolved by analysis beforehand, and having only obligations which must always be fulfilled, and which must be fulfilled immediately.

The Lithium language of Halpern and Weissman [HW03] has taken a more foundational approach by developing formalisms for policy that have well defined complexity results. However, they work in pure first-order logic which imposes on the policy author the burden of specifying complete definitions (so that every request has a decision) since one is not able to have default decision policies. As a simple example, assume that there is a policy that says that *only employees in the toy department are permitted to read file foo.* This can be encoded with the following two rules:

$$\forall X[dep(X, toy) \rightarrow permitted(X, read, foo)],$$
$$\forall X[\neg dep(X, toy) \rightarrow \neg permitted(X, read, foo)].$$

In addition we will need a complete definition specifying the members of the toy department. Let us say there are n members, named $p_1, \ldots, p_n$. We will then define the department as follows:

$$\forall X[X = p_1 \vee \cdots \vee X = p_n \leftrightarrow dep(X, toy)].$$

Besides the fact that this kind of formula will be needed in many situations, it is also true that each time that the composition of the toy department changes we will need to modify the formula. This is a well-known problem studied in the Knowledge Representation community, known as *elaboration tolerance.* 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. Take the following example from [HW03]:

$$\forall X[faculty(X) \rightarrow permitted(X, chair\_committees)],$$
$$\forall X[student(X) \rightarrow \neg permitted(X, chair\_committees)].$$

Here, faculty members are permitted to chair committees but students are not. Only when a faculty decides to take classes and become a student or a student is designated as faculty will a simple deductive analysis will find an inconsistency in this policy. In our analysis, unless the domain model restricts a subject to be either a student and faculty (but not both), we will be able to find out that there might be situations where this policy creates conflicts.

Irwin et al. propose a formalism for obligation policies together with analysis techniques [IYW06]. In the current paper, we have adapted the syntax of these

[7] See `http://ponder2.net/`.

obligation policies to produce a more general language that allows more complex policy rules to be expressed. However, the hierarchical structure of the rules in our language ensure that it is still computationally tractable, and that it 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 first order logic, such as stratified logic. Barker presents in [Bar00] a language that supports specification of access control policies using stratified clause-form logic, with emphasis on RBAC policies. However, this work does not discuss techniques for detecting conflicts in policy specifications. The Authorization Specification Language (ASL) [JSSB97], the Flexible Authorization Framework (FAF) [JSS97] and the extension to handle dynamic authorizations discussed in [CWJ04] are other examples of languages based on stratified clause-form logic. They also offer techniques for detecting modality conflicts and some application-specific conflicts in authorization policy specifications. One of the main differences between these pieces of work and ours is that they work with a fixed domain model. The model has users, groups of users and roles. Groups of users can be recursively collected into other groups to form a hierarchy. Roles are exactly like groups, i.e. recursive collections of groups of users. The difference appears when the authorizations assigned to groups and roles are applied to subjects. When a subject makes a request he has to pass a set of roles (the set could be empty). The first thing that is checked is that the subject actually belongs to these roles. Then the subject gets all the authorizations giving directly to him, all authorizations derived from his groups, and all the authorizations derived from the roles passed as parameters in the request. So the difference is that a user has no choice about the authorizations given based on groups, but can pick the roles (recall that authorizations can be positive or negative). In the language it is possible to write different policies for deriving authorizations based on roles and groups, and one can specify default policies like our availability rules and conflict resolution policies. By describing their domain model in our formalism, we can subsume their language in ours. However, fixing the domain model to roles and groups limits the kind of policies that they are able to express. For example, simple policies of the *three logging failures and you are out* type stand outside the expressive resources of these formalisms, but are very easy to express in ours. Another advantage of having control in the domain modeling is that we will able to model the administration of polices and reason about it. For example, our *req* inputs (and their consequent *do* actions) can add users to groups or roles, or create groups and roles. During analysis, we can then prove whether or not the system can get into a bad configuration which creates conflicts or violates application-dependent properties if there is a user with powerful rights but insufficient constraints.

8 Conclusion

We have presented a formal framework for the analysis of properties of an interesting class of authorization and obligation policies. We represent policies using a logical language, and join the rules expressing authorizations and obligations to other laws which describe the objects in, and behaviour of, the systems which the policies are intended to regulate. The dynamic system behaviour is represented using a variant of the Event Calculus. The whole formalism can be implemented as a normal logic program.

The purpose of this framework is to enable us to prove properties of our policies and the way they interact with the systems which they regulate. We see it as a significant benefit of our approach that it can detect under which system states and system histories modality conflicts (for instance, the joint presence of a permission and denial to perform a given action) arise, providing useful diagnostic information to the system designer, who may then decide that the given sequence of inputs is unlikely or not a substantial problem (and ignore the modality conflict as safe); or else decide to modify the policies and system behaviour so that the conflict, perceived as dangerous, will not arise.

We use a species of abductive constraint logic programming as the algorithm to perform our analysis; the output from this algorithm, in answer to a question about whether a given modality conflict, or separation-of-duty conflict, or other undesirable policy-related configuration arises, supplies just the kind of diagnostic information about the causes of the problem which we want. We described the essentials of the abductive approach we use, and gave sample analyses.

There are a number of outstanding themes and questions. We are exploring the possibility of incorporating different kinds of obligation policy rules in our formalism, so that instead of explict temporal arguments marking the beginning and end of the period during which an action ought to be performed, this period is dependent on the occurrence of events; the two approaches in defining obligations would be able to be mixed in our obligation policies. We see no reason why this should not be possible. We also wish to look, in greater detail, at more technical questions related to the underlying abductive constraint logic-programming technology used in our analysis algorithms, and to consider whether alternative approaches to abduction may yield increases in efficiency. We also wish to apply our analysis to a number of larger, real-world examples, to provide a more practical proof of the utility of our approach.

References

[Ban05] Arosha K Bandara. *A Formal Approach to Analysis and Refinement of Policies*. PhD thesis, Imperial College London, UK, July 2005.

[Bar00] S Barker. Security policy specification in logic. In *Proc. of Int. Conf. on Artificial Intelligence*, pages 143–148, June 2000.

[BFIK99] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[Cho95] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[CWJ04] Shiping Chen, Duminda Wijesekera, and Sushil Jajodia. Incorporating dynamic constraints in the flexible authorization framework. In *ESORICS*, pages 1–16, 2004.

[DFK06] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach

and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K.A. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, Washington, August 15-19 1988.

[Gol92] Robert Goldblatt. *Logics of time and computation*. Center for the Study of Language and Information, Stanford, CA, USA, 2nd edition, 1992.

[HW03] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *Proc. of 16th IEEE Computer Security Foundations Workshop*, page 187, 2003.

[IYW06] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In *Proc. of the 13th ACM Conf. on Computer and communications security*, pages 134–143, 2006.

[JSS97] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. of the IEEE Symposium on Security and Privacy*, page 31, 1997.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proc. of the ACM Int. SIGMOD Conf. on Management of Data*, May 1997.

[JSSS01] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[KMM00] Antonis C. Kakas, A. Michael, and Costas Mourlas. ACLP: Abductive constraint logic programming. *J. Log. Program.*, 44(1-3):129–177, 2000.

[KS86] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[LS01] Peter Loscocco and Stephen 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.

[LW06] Ninghui Li and Qihua Wang. Beyond separation of duty: an algebra for specifying high-level security policies. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 356–369, New York, NY, USA, 2006. ACM.

[MS02] Rob Miller and Murray Shanahan. Some alternative formulations of the event calculus. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 452–490. Springer, 2002.

[OAS05] OASIS XACML TC. extensible access control markup language (XACML) v2.0, 2005.

[RDD07] Giovanni Rusello, Changyu Dong, and Naranker Dulay. Authorisation and conflict resolution for hierarchical domains. In *Proc. of Int. Workshop on Policies for Distributed Systems and Networks*, June 2007.

[SYSR06] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 124–138, Washington, DC, USA, 2006. IEEE Computer Society.

[TN96] David Toman and Damian Niwinski. First-order queries over temporal databases inexpressible in temporal logic. In *Proc. of the 5th Int. Conf. on Extending Database Technology (EDBT)*, volume 1057, pages 307–324, 1996.

[VN04] Bert Van Nuffelen. *Abductive constraint logic programming: implementation and applications*. PhD thesis, K.U.Leuven, Belgium, June 2004.

A Test Results

A.1 The Project Meeting Example Revisited

We have built a prototype and tested the project meeting example described in Section 5 on the ASystem. ASystem is a free and open source abductive reasoning system developed by Van Nuffelen [VN04].

Apart from the information given in Section 5, we also have the following information about the projects and meetings:

- There are two projects – the *oakland* project led by Alice and the *ccs* project led by Bob.
- The oakland project has two project documents – *okl paper* and *okl slides*. The ccs project has two project documents – *ccs proposal* and *ccs report*.
- There is a meeting of the oakland project, scheduled to start at 50 hrs. Alice, Bob and Charles will be attending the meeting. The meeting documents are the *TO-DO* list and the meeting slides.
- Project leaders are always allowed to read meeting agendas before project meetings.
- Charles is told 5 hrs before the meeting to send the meeting slides to the project leader within 5 hrs after the meeting starts.
- Other people who might be involved in the two projects are Morris, Alessandra, Alberto, Emil and Rob.

All this information is reflected in the code in Section A.3.

A.2 Sample Testing

Here are some samples of the queries we made, and results obtained, using the ASystem with the code for the project meeting example presented in Appendix A.3.

Query 1 `Asystem ?- solve(permitted(Sub, Tar, read, T)).`

Given the code contained in Appendices A.3 and A.4, this query returned expected answers such as:

```
permitted(morris,okl_meeting,vote,1)
```

with the abduced:

```
initially(role(morris,standard_attender(okl_meeting)))
```

Also, answers such as:

```
permitted(alessandra,okl_meeting,view_votes,2)
```

with the abduced:

```
req(emil,alessandra,allocate(meeting_admin(okl_meeting)),0)
```

(The total set of answers is too large to show in full here.)

Query 2 `Asystem ?- solve(q(2, x(S, Ta, A, 1))).`

This query asks for a modality conflict (that between *permitted* and *denied* at time 1), as `q(2, X)` is defined in the code as

```
q(2, x(S, Ta, A, T)) :-
  permitted(S, Ta, A, T),
  denied(S, Ta, A, T).
```

The program showed that if

```
initially(role(morris,meeting_admin(okl_meeting)))
initially(role(emil,chair(okl_meeting)))
```

are abduced, then there is a modality conflict for requests by `emil` to perform the action `allocate(standard_attender(okl_meeting))` on `morris`. This is as we would expect, as in the circumstances given by the abduced initial state, the rules in Section 5 dictate both that this action will be *permitted* and *denied*. Many other circumstances leading to similar modality conflicts were found.

Query 3 `Asystem ?- solve(denied(Sub, Tar, Act, 1)).`

This query gave all of the possible answers. For example, one of them is:

```
denied(_,emil,allocate(standard_attender(okl_meeting)),1)
```

With the abduced atom:

```
req(alessandra,emil,allocate(meeting_admin(okl_meeting)),0)
```

Clearly, this is expected given that after such a request, `emil` will be fulfilling the role of the `meeting_admin` for the Oakland meeting, `okl_meeting`. In that circumstance, any request from someone to make `emil` a `standard_attender` of the same meeting will be denied, according to the negative authorization rule in the policy. Other expected solutions to this query are also found.

A.3 ASystem Code of the Project Meeting Example

In this section we present the particular, domain-specific code used in testing the 'project meeting' example from Section 5. The file, `project.pl`, first loads the general axioms presented later in Appendix A.4, and then includes the relevant policy rules and system behaviour specifications.

```
loadfile('pol.pl').

maxtime(15).

% -- defined predicates

defined(project(_)).
defined(meeting(_,_)).
defined(project_doc(_,_)).
defined(doc(_,_)).
defined(start_time(_,_)).
defined(attendee(_,_)).
defined(leader(_,_)).
defined(agenda(_, _)).
defined(sec_status(_)).
defined(role(_)).
defined(person(_)).

% -- signature

subject(Person) :-
  person(Person).

person(morris).
person(emil).
person(alessandra).
person(alberto).
person(rob).
person(alice).
person(bob).
person(charles).

target(M) :-
  meeting(M, _).
target(D) :-
  doc(D, _).
target(D) :-
  project_doc(D, _).
target(S) :-
  person(S).

action(read).
action(vote).
action(view_votes).
action(create).
action(classify(Status)) :-
  sec_status(Status).
action(allocate(Role)) :-
```

```
  role(Role).

sec_status(public).
sec_status(project).
sec_status(company).

role(meeting_admin(M)) :-
  meeting(M, _).
role(standard_attender(M)) :-
  meeting(M, _).
role(chair(M)) :-
  meeting(M, _).

fluent(role(S, R)) :-
  subject(S),
  role(R).

% -- values for the example

project(oakland).
project(ccs).

leader(alice, oakland).
leader(bob, ccs).

meeting(okl_meeting, oakland).
start_time(okl_meeting1, 30).

attendee(alice, okl_meeting1).
attendee(bob, okl_meeting1).
attendee(charles, okl_meeting1).

doc(m1_todos, okl_meeting1).
doc(m1_slides, okl_meeting1).

project_doc(okl_paper, oakland).
project_doc(okl_slides, oakland).

project_doc(ccs_proposal, ccs).
project_doc(ccs_report, ccs).

agenda(okl_meeting, okl_meeting_agenda).

% -- dynamic behaviour

initiates(S:Ta:allocate(R), role(Ta,R), T) :-
  person(S),
  person(Ta),
  role(R).

% -- positive authorization rules
%
% Project leaders are permitted to read
% meeting agendas for their projects prior
```

```
% to the start of the meeting.

permitted(S, A, read, T) :-
  time(T),
  project(P),
  leader(S, P),
  meeting(M, P),
  agenda(M, A),
  start_time(M, Ts),
  clp(T #< Ts).

% Project partners who are permitted to read
% the meeting agenda are allowed to read related
% project documentation 24 hours before the meeting
% starts.

permitted(S, D, read, T2) :-
  time(T2),
  project(P),
  meeting(M, P),
  agenda(M, A),
  permitted(S, A, read, T1),
  project_doc(D, P),
  start_time(M, T3),
  clp(T1 #< T2),
  clp(T2 #< T3),
  clp(T3 - T2 #< 24).

% Standard meeting attenders are allowed to vote at
% meetings; meeting administrators are allowed to
% view those votes. The meeting chair is allowed
% to allocate members of the company to these roles.
% Nobody is permitted to be both a standard meeting
% attender and a meeting admin.

permitted(S, M, vote, T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(S, standard_attender(M)), T).

permitted(S, M, view_votes, T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(S, meeting_admin(M)), T).

permitted(S, Ta, allocate(standard_attender(M)), T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(S, chair(M)), T).

permitted(S, Ta, allocate(meeting_admin(M)), T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(S, chair(M)), T).
```

```
denied(S, Ta, allocate(standard_attender(M)), T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(Ta, meeting_admin(M)), T).

denied(S, Ta, allocate(meeting_admin(M)), T) :-
  time(T),
  meeting(M, _),
  holdsAt(role(Ta, standard_attender(M)), T).

% A meeting attendee is not allowed to classify a
% document as 'public' if they created the document.

denied(S, D, classify(public), T2) :-
  time(T2),
  project(P),
  meeting(M, P),
  attendee(S, M),
  doc(D, M),
  do(S, D, create, T1),
  clp(T1 #< T2).

% Alice is allowed to create and declassify project documents,
% but not on the same project.

permitted(alice, D, create, T) :-
  time(T),
  project(P),
  not(denied(alice, D, create, T)).

permitted(alice, D, classify(public), T) :-
  time(T),
  project(P),
  project_doc(D, P),
  not(denied(alice, D, classify(public), T)).

denied(alice, D1, create, T2) :-
  time(T2),
  project(P),
  project_doc(D1, P),
  project_doc(D2, P),
  time(T1),
  clp(T1 #< T2),
  do(alice, D2, classify(public), T1).

denied(alice, D1, classify(public), T2) :-
  time(T2),
  project(P),
  project_doc(D1, P),
  project_doc(D2, P),
  time(T1),
  clp(T1 #< T2),
  do(alice, D2, create, T1).
```

```
% -- integrity constraints

ic :-
  req(S, S, _, _).

% -- queries

defined(q(_, _)).

q(1, x(S, Ta, A, T)) :-
  permitted(S, Ta, A, T).

q(2, x(S, Ta, A, T)) :-
  permitted(S, Ta, A, T),
  denied(S, Ta, A, T).

q(3, x(S, Ta, A, T)) :-
  denied(S, Ta, A, T).
```

A.4 ASystem Code for the General Axioms

In this section we give the code for the basic set-up used in all implementations of systems and policies using our framework in ASystem. The following is the code from the file `pol.pl`.

```
% ---------- abducible

abducible(req(_,_,_,_)).

% ---------- Defined Predicates

defined(permitted(_,_,_,_)).
defined(denied(_,_,_,_)).
defined(deny(_,_,_,_)).

defined(revoke(_,_,_,_,_,_)).

% -- the following are defined in the domain file

defined(subject(_)).
defined(target(_)).
defined(action(_)).
defined(dom_action(_)).
defined(fluent(_)).

% ---------- time
%
% maxtime/1 is defined in the domain file,
% for specific examples

defined(time(_)).
defined(times(_)).
defined(maxtime(_)).
```

```
time(T) :-
  maxtime(M),
  clp(T in 0..M).

times([]).

times([X|Rest]) :-
  time(X),
  times(Rest).

% ---------- event calculus

defined(holdsAt(_,_)).
defined(broken(_,_,_)).
defined(initiates(_,_,_)).
defined(terminates(_,_,_)).

abducible(initially(_)).
abducible(happens(_,_)).

holdsAt(F, T) :-
  initially(F),
  time(T),
  not(broken(F, 0, T)).

holdsAt(F, T) :-
  initiates(S:Ta:A, F, Ts),
  do(S, Ta, A, Ts),
  clp(Ts #< T),
  times([T,Ts]),
  not(broken(F, Ts, T)).

holdsAt(F, T) :-
  initiates(DomAct, F, Ts),
  happens(DomAct, Ts),
  clp(Ts #< T),
  times([Ts,T]),
  not(broken(F, Ts, T)).

broken(F, Ts, T) :-
  terminates(S:Ta:A, F, TT),
  do(S, Ta, A, TT),
  clp(Ts #< TT),
  clp(TT #< T),
  times([Ts,T,TT]).

broken(F, Ts, T) :-
  terminates(DomAct, F, TT),
  happens(DomAct, TT),
  clp(Ts #< TT),
  clp(TT #< T),
  times([Ts,T,TT]).
```

```
% ---------- interval predicates

defined(reqInBetween(_,_,_,_,_)).
defined(doInBetween(_,_,_,_,_)).

reqInBetween(Sub,Tar,Act,T1,T2) :-
  req(Sub,Tar,Act,Tm),
  clp(T1 #=< Tm),
  clp(Tm #< T2),
  times([T1,T2,Tm]).

doInBetween(Sub,Tar,Act,T1,T2) :-
  do(Sub,Tar,Act,Tm),
  clp(T1 #=< Tm),
  clp(Tm #< T2),
  times([T1,T2,Tm]).

% ---------- positive availability

defined(do(_,_,_,_)).

do(Sub, Tar, Act, T) :-
  req(Sub, Tar, Act, T),
  time(T),
  not(denied(Sub, Tar, Act, T)).

% ---------- Negative availability

deny(Sub, Tar, Act, T) :-
  req(Sub, Tar, Act, T),
  denied(Sub, Tar, Act, T).

% ---------- obligation

defined(obl(_,_,_,_,_,_)).
defined(cease_obl(_,_,_,_,_,_,_)).

cease_obl(Sub, Tar, Act, Tinit, Ts, Te, T) :-
  revoke(Sub, Tar, Act, Ts, Te, TT),
  clp(Tinit #=< TT),
  clp(TT #< T),
  clp(T #=< Te),
  times([Tinit,Te,T,TT]).

cease_obl(Sub, Tar, Act, Tinit, Ts, Te, T) :-
  do(Sub, Tar, Act, TT),
  clp(T1 #=< TT),
  clp(TT #< T),
  clp(T #=< Te),
  times([Tinit,Ts,Te,T,TT]).

% ---------- fulfilled

defined(fulfilled(_,_,_,_,_,_)).
```

```
fulfilled(Sub, Tar, Act, Ts, Te, T) :-
  obl(Sub, Tar, Act, Ts, Te, Tinit),
  do(Sub, Tar, Act, TT),
  not(cease_obl(Sub, Tar, Act, Tinit, Ts, Te, TT)),
  clp(Tinit #=< Ts),
  clp(Ts #=< TT),
  clp(TT #< Te),
  clp(TT #< T),
  time(T).

% ---------- violated

defined(violated(_,_,_,_,_,_)).

violated(Sub, Tar, Act, Ts, Te, T) :-
  obl(Sub, Tar, Act, Ts, Te, Tinit),
  not(cease_obl(Sub, Tar, Act, Tinit, Ts, Te, T2)),
  clp(Tinit #=< Ts),
  clp(Ts #< Te),
  clp(Te #=< T),
  time(T).

% ---------- domain-independent integrity constraints

% -- constraints for arguments for req/4

ic :-
  req(S, Ta, A, T),
  not(request_args(S,Ta,A)).

defined(request_args(_,_,_)).

request_args(S,Ta,A) :-
  subject(S),
  target(Ta),
  action(A).

% -- constraints for arguments to initially/1

ic :-
  initially(F),
  not(fluent(F)).

% -- constraints for arguments to happens/2

ic :-
  happens(A, T),
  not(dom_action(A)).
```