A Logic Based Approach for Dynamic Access Control

Vino Fernando Crescini and Yan Zhang

School of Computing and Information Technology
University of Western Sydney
Penrith South DC, NSW 1797, Australia
E-mail: {jcrescin,yan}@cit.uws.edu.au

Abstract. The *PolicyUpdater*[1] system is a fully-implemented access control system that provides policy evaluations as well as dynamic policy updates. These functions are achieved by the use of a logic-based language $\mathcal{L}$ to represent the underlying access control policies, constraints and update propositions. The system performs authorisation query evaluations and conditional policy updates by translating the language $\mathcal{L}$ to a normal logic program in a form suitable for evaluation using the *Stable Model* semantics.

1 Introduction

Recent advances in the information security field have produced a number of different approaches to access control, some of which are logic-based, e.g. [5, 7]. Bertino, et. al. [1] proposed an approach based on ordered logic with ordered domains. Jajodia, et. al. [6] on the other hand, proposed a general access control framework that features handling of multiple policies. However, these approaches lack the necessary details to address the issues involved in implementing a system based on these approaches.

The *Policy Description Language* or *PDL*, developed by Lobo, et. al. [8], is designed for representing event and action oriented generic policies. *PDL* was later extended by Chomicki, et. al. [3] to include a constraint mechanism called *policy monitors*. Bertino, et. al. [2], again took *PDL* a step further by extending *policy monitors* to support preferred constraints. While these languages possess enough expressive power to be used for most access control applications, systems based on these languages will not have the ability to perform dynamic policy updates.

To overcome these limitations, we propose the PolicyUpdater access control system. This system, with its own access control language, $\mathcal{L}$, allows policies to be represented as logical facts and rules with variable resolution and default propositions, and provides a mechanism to conditionally and dynamically perform a sequence of policy updates, as well as query evaluation.

The rest of this paper is organised as follows. In Section 2, the paper introduces language $\mathcal{L}$ with its formal syntax, semantics and some examples. Section 3 addresses the issues associated with domain consistency and query evaluation. Finally, Section 4 ends the paper with some concluding remarks.

[1] Web page at `http://www.cit.uws.edu.au/~jcrescin/projects/PolicyUpdater/index.html`

2 Language $\mathcal{L}$: Syntax and Semantics

Language $\mathcal{L}$ is a first-order logic language that is used to represent a policy base for an authorisation system. Two key features of the language are: (1) providing a means to conditionally and dynamically update the existing policy base and (2) having a mechanism by which queries may be evaluated from the updated policy base.

2.1 Syntax

Logic programs of language $\mathcal{L}$ are composed of language statements, each terminated by a semicolon ";" character. Comments may appear anywhere in the logic program and, like C, language $\mathcal{L}$ comments are delimited by the "/*" and "*/"

Components of Language $\mathcal{L}$

Identifiers. The most basic unit of language $\mathcal{L}$ is the identifier. Identifiers are used to represent different components of the language, and is defined as an upper or lower case alphabet character, followed by 0 to 127 characters of alphabet, digit or underscore characters. There are 3 types of identifiers, each defined by the following syntax:

```
[a-zA-Z]([a-zA-Z0-9_]){0,127}
```

- *Entity Identifiers* represent constant entities that make up a logical atom. They are divided further into 3 types, with each type again divided into the *singular entity* and *group entity* categories: *Subjects* (e.g. alice, lecturers); *Access Rights* (e.g. read, write, own); and *Objects* (e.g. file, database, directory). This type of identifier must start with a lowercase character.
- *Policy Update Identifiers* are used for the sole purpose of naming a policy update. These identifier names are then used as labels to refer to policy update definitions and directives. As labels, identifiers of this class occupy a different namespace from entity identifiers. For this reason, policy update identifiers share the same syntax with entity identifiers.
- *Variable Identifiers* are used as entity identifier place-holders. To distinguish them from entity and policy update identifiers, variable identifiers are prefixed with an upper-case character.

Atoms. An atom is composed of a relation with 2 to 3 entity or variable identifiers that represent a logical relationship between the entities. There are 3 types of atoms:

- *Holds.* An atom of this type states that the subject identifier sub holds the access right identifier acc for the object identifier obj.

  ```
  holds(<sub>, <acc>, <obj>)
  ```

- *Membership.* This type of atom states that the singular identifier elt is a member or element of the group identifier grp. It is important to note that identifiers elt and grp must be of the same base type (e.g. subject and subject group).

```
memb(<elt>, <grp>)
```

– *Subset.* The subset atom states that the group identifiers $grp1$ and $grp2$ are of the same types and that group $grp1$ is a subset of the group $grp2$.

```
subst(<grp1>, <grp2>)
```

Facts. A fact makes a claim that the relationship represented by an atom or its negation holds in the current context. Facts are negated by the use of the negation operator "!". The following shows the formal syntax of a fact:

```
[!]<holds_atom>|<memb_atom>|<subst_atom>
```

Expressions. An expression is either a fact, or a logical conjunction of facts, separated by the double-ampersand characters "$\&\&$".

```
<fact1> [&& <fact2> [&& ...]]
```

Atoms that contain no variables, i.e. composed entirely of entity identifiers, are called *ground atoms*. Similarly, facts and expressions composed of ground atoms are called *ground facts* and *ground expressions*, respectively.

Definition Statements

Entity Identifier Definition. All entity identifiers (subjects, access rights, objects and groups) must first be declared before any other statements to define the entity domain of the policy base. The following entity declaration syntax illustrates how to define one or more entity identifiers of a particular type.

```
ident sub|acc|obj[-grp] <ent_id>[, ...];
```

Initial Fact Definition. The initial facts of the policy base, those that hold before any policy updates are performed, are defined by using the following definition syntax:

```
initially <ground_exp>;
```

Constraint Definition. Constraints are logical rules that hold regardless of any changes that may occur when the policy base is updated. The constraint rules are true in the initial state and remain true even after a policy update is performed.

The constraint syntax below shows that for any state of the policy base, expression $ex1$ holds if expression $ex2$ is true and there is no evidence that $ex3$ is true. The $with\ absence$ clause allows constraints to behave like default propositions, where the absence of proof that an expression holds satisfies the clause condition of the proposition.

It is important to note that the expressions $ex1$, $ex2$ and $ex3$ may be non-ground expressions, which allows identifiers occurring in these expressions to be variables.

```
always <ex1> [implied by <ex2> [with absence <ex3>]];
```

Policy Update Definition. Before a policy update can be applied, it must first be defined by using the following syntax:

```
<up_id>([<var_id>[, ...]]) causes <ex1> if <ex2>;
```

up_id is the policy update identifier to be used in referencing this policy update. The optional var_id list are the variable identifiers occurring in the expressions $ex1$ and $ex2$ and will eventually be replaced by entity identifiers when the update is referenced. The postcondition expression $ex1$ is an expression that will hold in the state after this update is applied. The expression $ex2$ is a precondition expression that must hold in the current state before this update is applied.

Note that a policy update definition will have no effect on the policy base until it is applied by one of the directives described in the following section.

Directive Statements

Policy Update Directives. The policy update sequence list contains a list of references to defined policy updates in the domain. The policy updates in the sequence list are applied to the current state of the policy base one at a time to produce a policy base state upon which queries can be evaluated. The following four directives are the policy sequence manipulation features of language $\mathcal{L}$.

Adding an update into the sequence. Defined policy updates are added into the sequence list through the use of the following directive:

```
seq add <up_id>([<ent_id>[, ...]]);
```

where up_id is the identifier of a defined policy update and the ent_id list is a comma-separated list of entity identifiers that will replace the variable identifiers that occur in the definition of the policy update.

Listing the updates in the sequence. The following directive may be used to list the current contents of the policy update sequence list.

```
seq list;
```

This directive is answered with an ordinal list of policy updates in the form:

```
<n> <up_id>([<ent_id>[, ...]])
```

where n is the ordinal index of the policy update within the sequence list starting at 0. up_id is the policy update identifier and the ent_id list is the list of entity identifiers used to replace the variable identifier place-holders.

Removing an update from the sequence. The syntax below shows the directive to remove a policy update reference from the list. n is the ordinal index of the policy update to be removed. Note that removing a policy update reference from the sequence list may change the ordinal index of other update references.

```
seq del <n>;
```

Computing an update sequence. The policy updates in the sequence list is not applied until the $compute$ directive is issued. The directive causes the policy update references in the sequence list to be applied one at a time in the same order that they appear in the list. The directive also causes the system to generate the policy base models against which query requests can be evaluated.

```
compute;
```

Query Directive. A ground query expression may be issued against the current state of the policy base. This current state is derived after all the updates in the update sequence have been applied, one at a time, upon the initial state. Query expressions are answered with a $true$, $false$ or an $unknown$, depending on whether the queried expression holds, its negation holds, or neither, respectively. Syntax is as follows:

```
query <ground_exp>;
```

Example 1 The following language $\mathcal{L}$ program code listing shows a simple rule-based document access control system scenario.

In this example, the subject $alice$ is initially a member of the subject group $grp2$, which is a subset of group $grp1$. The group $grp1$ also initially holds a $read$ access right for the object $file$. The constraint states that if the group $grp1$ has $read$ access for $file$, and no other information is present to conclude that $grp3$ do not have $write$ access for $file$, then the group $grp1$ is granted $write$ access for $file$. For simplicity, we only consider one policy update $delete\_read$ and a few queries that are evaluated after the policy update is performed.

```
ident sub alice;
ident sub-grp grp1, grp2, grp3;
ident acc read, write;
ident obj file;

initially memb(alice, grp2) && subst(grp2, grp1);
initially holds(grp1, read, file);

always holds(grp1, write, file)
  implied by holds(grp1, read, file)
  with absence !holds(grp3, write, file);

delete_read(SG0, OS0) causes !holds(SG0, read, OS0);

seq add delete_read(grp1, file);

compute;

query holds(grp1, write, file);
query holds(alice, read, file);
```

■

2.2 Semantics

After giving a detailed syntactic definition of language $\mathcal{L}$, we now define its formal semantics.

Domain Description of Language $\mathcal{L}$

Definition 1 *The domain description $\mathcal{D}_{\mathcal{L}}$ of language $\mathcal{L}$ is defined as a finite set of ground initial state facts, constraint rules and policy update definitions.*

In addition to the domain description $\mathcal{D}_{\mathcal{L}}$, language $\mathcal{L}$ also includes an additional set: the sequence list ψ. The sequence list ψ is an ordered set that contains a sequence of references to policy update definitions. Each policy update reference consists of the policy update identifier and a series of zero or more identifier entities to replace the variable place-holders in the policy update definitions.

Language $\mathcal{L}^*$ In language $\mathcal{L}$, the policy base is subject to change, which is triggered by the application of policy updates. Such changes bring forth the concept of policy base states. Conceptually, a state may be thought of as a set of facts and constraints of the policy base at a particular instant. The state transition notation $PB \overrightarrow{u} PB'$ shows that a new state PB' is generated from the current state PB after the policy update u is applied.

This concept of a state means that for every policy update applied to the policy base, a new instance of the policy base or a new set of facts and constraints are generated. To precisely define the underlying semantics of domain description $\mathcal{D}_{\mathcal{L}}$ in language $\mathcal{L}$, we introduce language $\mathcal{L}^*$, which is an extended logic program representation of language $\mathcal{L}$, with state as an explicit sort.

Language $\mathcal{L}^*$ contains only one special state constant S_0 to represent the initial state of a given domain description. All other states are represented as a resulting state obtained by applying the Res function.

The $Res(u, \sigma)$ function takes a policy update reference u, where $u \in \psi$, and the current state σ as input arguments and returns the resulting state after update u has been applied to state σ. Given an initial state S_0 and a sequence list ψ, each state σ_i ($0 \leq i \leq |\psi|$) may be represented as $\sigma_0 = S_0$, $\sigma_1 = Res(u_0, \sigma_0)$, ..., $\sigma_{|\psi|} = Res(u_{|\psi|-1}, \sigma_{|\psi|-1})$. Substituting each state with a recursive call to the Res function, the final state $S_{|\psi|}$ is defined as $S_{|\psi|} = Res(u_{|\psi|-1}, Res(\ldots, Res(u_0, S_0)))$.

Entities. The entity set $\mathcal{E}$ is a union of six disjoint entity sets: single subject $\mathcal{E}_{ss}$, group subject $\mathcal{E}_{sg}$, single access right $\mathcal{E}_{as}$, group access right $\mathcal{E}_{ag}$, single object $\mathcal{E}_{os}$ and group object $\mathcal{E}_{og}$. We also define three additional entity sets: $\mathcal{E}_s$, $\mathcal{E}_a$ and $\mathcal{E}_o$, which are unions of their respective singular and group entity sets. Each entity in set $\mathcal{E}$ corresponds directly to the *entity identifiers* of language $\mathcal{L}$.

Atoms. The main difference between language $\mathcal{L}$ and language $\mathcal{L}^*$ lies in the definition of an atom. Atoms in language $\mathcal{L}^*$ represent a logical relationship of 2 to 3 entities in a particular state. That is, language $\mathcal{L}^*$ atoms have an extra parameter to specify the

state in which they hold. In this paper, atoms of language $\mathcal{L}^*$ are written with the hat character ($\hat{holds}$, $\hat{memb}$ and $\hat{subst}$) to differentiate from the atoms of language $\mathcal{L}$. The atom set $\mathcal{A}^\sigma$ is the set of all atoms in state σ.

Facts. In language $\mathcal{L}^*$, a fact states whether an atom or its negation holds in a particular state. A fact f in state σ is formally defined as $f^\sigma = [\neg]\alpha, \alpha \in \mathcal{A}^\sigma$.

Translating Language $\mathcal{L}$ to Language $\mathcal{L}^*$ Given a domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$, we translate $\mathcal{D}_\mathcal{L}$ into an extended logic program of language $\mathcal{L}^*$, as denoted by $Trans(\mathcal{D}_\mathcal{L})$. The semantics of $\mathcal{D}_\mathcal{L}$ is provided by the answer sets of program $Trans(\mathcal{D}_\mathcal{L})$. Before we can fully define $Trans(\mathcal{D}_\mathcal{L})$, we must first define the following functions:

The $CopyAtom()$ function takes two arguments: an atom of language $\mathcal{L}^*$ at some state and new state. The function returns an equivalent atom of the same type and with the same entities, but in the new state specified.

Another function, $TransAtom()$, takes an atom α of language $\mathcal{L}$ and an arbitrary state σ. It then returns a language $\mathcal{L}^*$ atom of the same type in state σ, with the same given entities. The other function, $TransFact()$, is similar to the $TransAtom()$ function, but instead of translating an atom, it takes a fact from language $\mathcal{L}$ and a state then returns the equivalent fact in language $\mathcal{L}^*$.

Initial Fact Rules. Translating initial fact expressions of language $\mathcal{L}$ to language $\mathcal{L}^*$ rules is a trivial procedure: translate each fact that make up the initial fact expression of language $\mathcal{L}$ with its corresponding equivalent initial state atom of language $\mathcal{L}^*$. For example, the following code shows a language $\mathcal{L}$ $initially$ statement:

```
initially holds(bob, read, file) && memb(alice, users);
```

in language $\mathcal{L}^*$, the above statement is translated to:

$\hat{holds}(bob, read, file, S_0) \leftarrow$
$\hat{memb}(alice, users, S_0) \leftarrow$

Constraint Rules. Each constraint rule in language $\mathcal{L}$ is expressed as a series of logical rules in language $\mathcal{L}^*$. Given that all variable occurrences have been grounded to entity identifiers, a constraint in language $\mathcal{L}$, with $m, n, o \geq 0$ may be represented as:

```
always a_0 && ... && a_m
  implied by b_0 && ... && b_n
  with absence c_0 && ... && c_o;
```

Each fact in the $always$ clause of a language $\mathcal{L}$ constraint corresponds to a new rule, where it is the consequent. Each of these new rules will have expression b in the $implied\ by$ clause as the positive premise and the expression c in the $with\ absence$ clause as the negative premise.

$\hat{a_0} \leftarrow \hat{b_0}, \ldots, \hat{b_n}, not\ \hat{c_0}, \ldots, not\ \hat{c_o}$
$\ldots$
$\hat{a_m} \leftarrow \hat{b_0}, \ldots, \hat{b_n}, not\ \hat{c_0}, \ldots, not\ \hat{c_o}$

For example, given a policy update reference in the sequence list ψ (i.e. $|\psi| = 1$) and the following language $\mathcal{L}$ code fragment:

```
always holds(bob, read, f1) && holds(bob, write, f1)
  implied by memb(bob, grp)
  with absence !holds(bob, own, f1);
```

The following shows the language $\mathcal{L}^*$ translation:

$\hat{holds}(bob, read, f1, S_0) \leftarrow \hat{memb}(bob, grp, S_0), not \neg \hat{holds}(bob, own, f1, S_0)$
$\hat{holds}(bob, write, f1, S_0) \leftarrow \hat{memb}(bob, grp, S_0), not \neg \hat{holds}(bob, own, f1, S_0)$
$\hat{holds}(bob, read, f1, S_1) \leftarrow \hat{memb}(bob, grp, S_1), not \neg \hat{holds}(bob, own, f1, S_1)$
$\hat{holds}(bob, write, f1, S_1) \leftarrow \hat{memb}(bob, grp, S_1), not \neg \hat{holds}(bob, own, f1, S_1)$

Policy Update Rules. With all occurrences of variable place-holders grounded to entity identifiers, a language $\mathcal{L}$ policy update can then be translated to language $\mathcal{L}^*$. In language $\mathcal{L}^*$, policy updates are represented as a set of implications, with each fact in the postcondition expression as the consequent and precondition expression as the premise. However, the translation process must also take into account that the premise of the implication holds in the state before the policy update is applied and that the consequent holds in the state after the application [10]. For example, given an update sequence list ψ = $\{grant\_read, grant\_write\}$ and the following language $\mathcal{L}$ policy update definitions:

```
grant_read()
  causes holds(bob, read, file) if memb(bob, readers);
grant_write()
  causes holds(bob, write, file) if memb(bob, writers);
```

The following shows the language $\mathcal{L}^*$ translation:

$\hat{holds}(bob, read, file, S_1) \leftarrow \hat{memb}(bob, readers, S_0)$
$\hat{holds}(bob, write, file, S_2) \leftarrow \hat{memb}(bob, writers, S_1)$

Inheritance Rules. All properties held by a group are inherited by all the members and subsets of that group. This rule is easy to apply for subject group entities. However, careful attention must be given to access right and object groups. A subject holding an access right for an object group implies that the subject also holds that access right for all objects in the object group. Similarly, a subject holding an access right group for a particular object implies that the subject holds all access rights contained in the access right group for that object.

A conflict is encountered when a particular property is to be inherited by an entity from a group of which it is a member or subset, and the contained entity already holds

the negation of that property. This conflict is resolved by giving negative facts higher precedence over its positive counterpart: by allowing member or subset entities to inherit its parent group's properties only if the entities do not already hold the negation of those properties.

The following are the inheritance constraint rules to allow the properties held by a subject group to propagate to all of its members that do not already hold the negation of the properties. For all s_s, s_g, a, o, σ where $s_s \in \mathcal{E}_{ss}$, $s_g \in \mathcal{E}_{sg}$, $a \in \mathcal{E}_a$, $o \in \mathcal{E}_o$ and $S_0 \leq \sigma \leq S_{|\psi|}$:

$$\hat{holds}(s_s, a, o, \sigma) \leftarrow \hat{holds}(s_g, a, o, \sigma), \hat{memb}(s_s, s_g, \sigma), not \neg \hat{holds}(s_s, a, o, \sigma)$$
$$\neg \hat{holds}(s_s, a, o, \sigma) \leftarrow \neg \hat{holds}(s_g, a, o, \sigma), \hat{memb}(s_s, s_g, \sigma)$$

The rules below represent inheritance rules for subject groups to allow subsets to inherit properties held by their supergroup. Note that there is also a set of corresponding rules to represent membership and subset inheritance for access right and object groups. For all s_{g1}, s_{g2}, a, o, σ where s_{g1}, $s_{g2} \in \mathcal{E}_{sg}$, $a \in \mathcal{E}_a$, $o \in \mathcal{E}_o$ and $S_0 \leq \sigma \leq S_{|\psi|}$:

$$\hat{holds}(s_{g1}, a, o, \sigma) \leftarrow$$
$$\hat{holds}(s_{g2}, a, o, \sigma), \hat{subst}(s_{g1}, s_{g2}, \sigma), not \neg \hat{holds}(s_{g1}, a, o, \sigma)$$
$$\neg \hat{holds}(s_{g1}, a, o, \sigma) \leftarrow \neg \hat{holds}(s_{g2}, a, o, \sigma), \hat{subst}(s_{g1}, s_{g2}, \sigma)$$

Transitivity Rules. Given three group entities G, G' and G''. If G is a subset of G' and G' is a subset of G'', then G must also be a subset of G''. The following rules ensure that the transitive property holds for subject groups. Note that similar rules exist to ensure that the transitive property also holds for access right and object groups. For all s_{g1}, s_{g2}, s_{g3}, σ where sg_1, sg_2, $sg_3 \in \mathcal{E}_{sg}$ and $S_0 \leq \sigma \leq S_{|\psi|}$:

$$\hat{subst}(s_{g1}, s_{g3}, \sigma) \leftarrow \hat{subst}(s_{g1}, s_{g2}, \sigma), \hat{subst}(s_{g2}, s_{g3}, \sigma)$$

Inertial Rules. Intuitively, all facts in the current state that are not affected by a policy update should be carried over to the next state after the update. In language $\mathcal{L}^*$, this rule must be explicitly stated as a constraint. Formally, the inertial rules are expressed as follows. For all $\hat{\alpha}$, u, there is an $\hat{\alpha}'$ where $\hat{\alpha} \in \mathcal{A}^\sigma$, $u \in \psi$ and $\hat{\alpha}' = CopyAtom(\hat{\alpha}, Res(u, \sigma))$:

$$\hat{\alpha}' \leftarrow \hat{\alpha}, not \neg \hat{\alpha}'$$
$$\neg \hat{\alpha}' \leftarrow \neg \hat{\alpha}, not \hat{\alpha}'$$

Definition 2 *Given a domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$, its language $\mathcal{L}^*$ translation $Trans(\mathcal{D}_\mathcal{L})$ is an extended logic program of language $\mathcal{L}$ consisting of: (1) initial fact rules, (2) constraint rules, (3) policy update rules, (4) inheritance rules, (5) transitivity rules, and (6) inertial rules, as described above.*

By using the above definition, we can now state a theorem that defines the maximum number of rules generated in a translation $Trans(\mathcal{D}_\mathcal{L})$ given a domain description $\mathcal{D}_\mathcal{L}$. With this theorem, we show that the size of the translated domain $|Trans(\mathcal{D}_\mathcal{L})|$ is only polynomially larger than the size of the given domain $|\mathcal{D}_\mathcal{L}|$.

Theorem 1 (Translation Size[2]) *Given a domain description $\mathcal{D}_\mathcal{L}$; the sets $\mathcal{S}_i$, $\mathcal{S}_c$ and $\mathcal{S}_u$ containing the initially, constraint and policy update statements in $\mathcal{D}_\mathcal{L}$, respectively; the set of all entities $\mathcal{E}$ in $\mathcal{D}_\mathcal{L}$, including its subsets $\mathcal{E}_s$, $\mathcal{E}_a$, $\mathcal{E}_s$, $\mathcal{E}_{ss}$, $\mathcal{E}_{as}$, $\mathcal{E}_{os}$, $\mathcal{E}_{sg}$, $\mathcal{E}_{ag}$, $\mathcal{E}_{og}$; the set $\mathcal{A}$ containing all the atoms in $\mathcal{D}_\mathcal{L}$; the maximum number of facts M_i in any statement in $\mathcal{S}_i$; the maximum number of facts M_c in the $always$ clause of any statement in $\mathcal{S}_c$; the maximum number of facts M_u in the postcondition of any statement in $\mathcal{S}_u$; and finally the sequence list ψ.*

$$
\begin{aligned}
&|Trans(\mathcal{D}_\mathcal{L})| = \\
&\quad M_i\,|\mathcal{S}_i|\,+\,M_c\,|\mathcal{S}_c|\,|\psi|\,+\,M_u\,|\psi|\,+\,|\psi|\,(|\mathcal{E}_{sg}|^3\,+\,|\mathcal{E}_{ag}|^3\,+\,|\mathcal{E}_{og}|^3)\,+\,2\,|\mathcal{A}|\,|\psi|\,+ \\
&\quad 2|\psi|\,(\,|\mathcal{E}_{ss}|\,|\mathcal{E}_{sg}|\,|\mathcal{E}_a|\,|\mathcal{E}_o|\,+\,|\mathcal{E}_s|\,|\mathcal{E}_{as}|\,|\mathcal{E}_{ag}|\,|\mathcal{E}_o|\,+\,|\mathcal{E}_s|\,|\mathcal{E}_a|\,|\mathcal{E}_{os}|\,|\mathcal{E}_{og}|\,)\,+ \\
&\quad 2|\psi|\,(\,|\mathcal{E}_{sg}|^2\,|\mathcal{E}_a|\,|\mathcal{E}_o|\,+\,|\mathcal{E}_s|\,|\mathcal{E}_{ag}|^2\,|\mathcal{E}_o|\,+\,|\mathcal{E}_s|\,|\mathcal{E}_a|\,|\mathcal{E}_{og}|^2\,)
\end{aligned}
$$

3 Domain Consistency Checking and Evaluation

A domain description of language $\mathcal{L}$ must be consistent in order generate a consistent answer set for the evaluation of queries. This section considers two issues: the problem of identifying whether a given domain description is consistent, and how query evaluation is performed given a consistent language domain description. Before these issues can be considered, a few notational constructs must first be introduced. Given a domain description $\mathcal{D}_\mathcal{L}$ composed of the following language $\mathcal{L}$ statements:

```
initially a_0 && ... && a_m && !b_0 && ... && !b_n;
always c_0 && ... && c_o && !d_0 && ... && !d_p
  implied by e_0 && ... && e_q && !f_0 && ... && !f_r
  with absence g_0 && ... && g_s && !h_0 && ... && !h_t;
update()
  causes i_0 && ... && i_u && !j_0 && ... && !j_v
  if k_0 && ... && k_w && !l_0 && ... && !l_x;
```

We define the 6 sets of ground facts:

$\mathcal{F}^+_{int} = \{a_z \mid 0 \leq z \leq m\}$, $\mathcal{F}^+_{con} = \{c_z \mid 0 \leq z \leq o\}$, $\mathcal{F}^+_{upd} = \{i_z \mid 0 \leq z \leq u\}$,
$\mathcal{F}^-_{int} = \{b_z \mid 0 \leq z \leq n\}$, $\mathcal{F}^-_{con} = \{d_z \mid 0 \leq z \leq p\}$, $\mathcal{F}^-_{upd} = \{j_z \mid 0 \leq z \leq v\}$

Additionally, we use the complementary set notation $\overline{\mathcal{F}}$ to denote a set containing the negation of facts in set $\mathcal{F}$, i.e. $\overline{\mathcal{F}} = \{\neg\rho \mid \rho \in \mathcal{F}\}$. Furthermore, we define the following functions. Let γ be an initial, constraint or policy update definition statement of language $\mathcal{L}$:

$$
Eff(\gamma) = \begin{cases} \{a_0, \ldots, a_m, \neg b_0, \ldots, \neg b_n\}, \text{ if } \gamma \text{ is an initially statement} \\ \{c_0, \ldots, c_o, \neg d_0, \ldots, \neg d_p\}, \text{ if } \gamma \text{ is a constraint statement} \\ \{i_0, \ldots, i_u, \neg j_0, \ldots, \neg j_v\}, \text{ if } \gamma \text{ is a policy update statement} \end{cases}
$$

$$
Def(\gamma) = \begin{cases} \emptyset, \text{ if } \gamma \text{ is an initially statement} \\ \{g_0, \ldots, g_s, \neg h_0, \ldots, \neg h_t\}, \text{ if } \gamma \text{ is a constraint statement} \\ \emptyset, \text{ if } \gamma \text{ is a policy update statement} \end{cases}
$$

$$Pre(\gamma) = \begin{cases} \emptyset, \text{ if } \gamma \text{ is an initially statement} \\ \{e_0, \ldots, e_q, \neg f_0, \ldots, \neg f_r\}, \text{ if } \gamma \text{ is a constraint statement} \\ \{k_0, \ldots, k_w, \neg l_0, \ldots, \neg l_x\}, \text{ if } \gamma \text{ is a policy update statement} \end{cases}$$

Definition 3 *Given a domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$, two ground facts ρ and ρ' are* mutually exclusive *in $\mathcal{D}_\mathcal{L}$ if:*

$\rho \in \{\mathcal{F}^+_{int} \cup \overline{\mathcal{F}^-_{int}} \cup \mathcal{F}^+_{con} \cup \overline{\mathcal{F}^-_{con}} \cup \mathcal{F}^+_{upd} \cup \overline{\mathcal{F}^-_{upd}}\}$ *implies*
$\rho' \notin \{\mathcal{F}^+_{int} \cup \overline{\mathcal{F}^-_{int}} \cup \mathcal{F}^+_{con} \cup \overline{\mathcal{F}^-_{con}} \cup \mathcal{F}^+_{upd} \cup \overline{\mathcal{F}^-_{upd}}\}$

Simply stated, a pair of mutually exclusive facts cannot both be true in any given state. The following two definitions refer to language $\mathcal{L}$ statements.

Definition 4 *Given a domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$, two statements γ and γ' are* complementary *in $\mathcal{D}_\mathcal{L}$ if one of the following conditions holds:*

1. *γ and γ' are both constraint statements and $Eff(\gamma) = \overline{Eff(\gamma')}$.*
2. *γ is a constraint statement, γ' is an update statement and $Eff(\gamma) = \overline{Eff(\gamma')}$.*

Definition 5 *Given a domain description $\mathcal{D}_\mathcal{L}$, $\mathcal{D}_\mathcal{L}$ is said to be* normal *if it satisfies all of the following conditions:*

1. *$\mathcal{F}^+_{int} \cap \mathcal{F}^-_{int} = \emptyset$*
2. *For all constraint statements γ in $\mathcal{D}_\mathcal{L}$, $\overline{Eff(\gamma)} \cap Pre(\gamma) = \emptyset$.*
3. *For any two constraint statements γ and γ' in $\mathcal{D}_\mathcal{L}$, $Def(\gamma) \cap Eff(\gamma') = \emptyset$.*
4. *For any two* complementary *statements γ and γ' in $\mathcal{D}_\mathcal{L}$, there exists a pair of ground expression $\epsilon \in Pre(\gamma)$ and $\epsilon' \in Pre(\gamma')$ such that ϵ and ϵ' are* mutually exclusive.

With the above definitions, we can now provide a sufficient condition to ensure the consistency of a domain description.

Theorem 2 (Domain Consistency[2]) *A* normal *domain description of language $\mathcal{L}$ is also* consistent.

Basically, only consistent domain descriptions can be evaluated in terms of user queries. For this reason, Theorem 2 may be used to check whether a domain description is consistent.

Definition 6 *Given a* consistent *domain description $\mathcal{D}_\mathcal{L}$, ground query expression ϕ and a finite sequence list ψ, we say query ϕ holds in $\mathcal{D}_\mathcal{L}$ after all policy updates the in sequence list ψ have been applied, denoted as $\mathcal{D}_\mathcal{L} \models \{\phi, \psi\}$, if and only if for every fact $\rho \in \phi$, $TransFact(\rho, S_{|\psi|})$ is in every answer set of $Trans(\mathcal{D}_\mathcal{L})$.*

Definition 6 shows that given a finite list of policy updates ψ, a query expression ϕ may be evaluated from a consistent language $\mathcal{L}$ domain $\mathcal{D}_\mathcal{L}$. This is achieved by generating a set of answer sets from the normal logic program translation $Trans(\mathcal{D}_\mathcal{L})$. ϕ is then said to hold in $\mathcal{D}_\mathcal{L}$ after the policy updates in ψ have been applied if and only if every answer set generated contains every fact in the query expression ϕ.

[2] The proof of these theorems are presented in the full version of this paper [4].

Example 2 Given the language $\mathcal{L}$ program listing in Example 1 and the sequence list $\psi = \{delete\_read(grp1, file)\}$. The following shows the results of each query ϕ:

$\phi_0 = holds(grp1, write, file)$: TRUE
$\phi_1 = holds(alice, read, file)$: FALSE

■

4 Conclusion

In this paper, we have presented the PolicyUpdater system, a logic-based authorisation system that features query evaluation and dynamic policy updates. This is made possible by the use of a first-order logic language, $\mathcal{L}$, for defining, updating and querying of access control policies. As we have shown, language $\mathcal{L}$ is expressive enough to represent constraints and default rules. The full PolicyUpdater system implementation is presented in [4].

One possible future extension to this work is to integrate temporal logic in language $\mathcal{L}$ to allow temporal constraints to be expressed in access control policies. This extension will be useful in e-commerce applications where authorisations are granted or denied based on time dependent policies.

References

1. Bertino, E., Buccafurri, F., Ferrari, E., Rullo, P., A Logic-based Approach for Enforcing Access Control. *Journal of Computer Security*, Vol. 8, No. 2-3, pp. 109-140, IOS Press, 2000.
2. Bertino, E., Mileo A., Provetti, A., Policy Monitoring with User-Preferences in PDL. In *Proceedings of IJCAI-03 Workshop for Nonmonotonic Reasoning, Action and Change*, pp. 37-44, 2003.
3. Chomicki, J., Lobo, J., Naqvi S., A Logic Programming Approach to Conflict Resolution in Policy Management. In *Proceedings of KR2000, 7th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 121-132, Kaufmann, 2000.
4. Crescini, V. F., Zhang, Y., *PolicyUpdater - A System for Dynamic Access Control*. 2004 (manuscript).
5. Halpern, J. Y., Weissman, V., Using First-Order Logic to Reason About Policies. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pp.187-201, 2003.
6. Jajodia, S., Samarati, P., Sapino, M. L., Subrahmanian, V. S., Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, Vol. 29, No. 2, pp. 214-260, 2001.
7. Li, N., Grosof, B. N., Feigenbaum, J., Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM Transactions on Information and System Security*, Vol. 6, No. 1, pp. 128-171, 2003.
8. Lobo, J., Bhatia, R., Naqvi, S., A Policy Description Language. In *Proceedings of AAAI 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence*, pp. 291-298, AAAI Press, 1999.
9. Simons, P., Efficient Implementation of the Stable Model Semantics for Normal Logic Programs. *Research Reports, Helsinki University of Technology*, No. 35, 1995.
10. Zhang, Y., Logic Program Based Updates. *ACM Transactions on Computational Logic*, 2004 (to appear).