International Journal of Information Security manuscript No.
(will be inserted by the editor)

Vino Fernando Crescini · Yan Zhang

PolicyUpdater – A System for Dynamic Access Control

Abstract *PolicyUpdater* is a fully-implemented authorisation 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 access control query evaluations and conditional policy updates by translating the language $\mathcal{L}$ policies to a normal logic program in a form suitable for evaluation using the *Stable Model* semantics. In this paper, we show the underlying mechanisms that make up the PolicyUpdater system, including the theoretical foundation of its formal language, system structure, implementation issues and some performance analysis.

Keywords access control · authorisation · artificial intelligence · logic programming · policy update

1 Introduction

The traditional access control mechanism is the *Access Control Matrix* where columns represent subjects, rows represent objects and each cell contains the access-rights of a subject over a particular object. However, flexibility and scalability issues arise when such method is used on real-world applications. A more effective access control paradigm is the logic-based approach. In this approach, instead of explicitly defining all access-rights of all subjects for all objects in a domain, a set of logical facts and rules are used to define the policy base.

Recent advances in the field have produced a number of different approaches to logic-based access control systems, e.g. [12, 16]. One such access control system was proposed by Bai and Varadharajan [3, 4]. Their system's key characteristic is its ability to dynamically update an otherwise static policy base.

V. F. Crescini · Y. 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

Another system, proposed by Bertino, et. al. [6], uses an authorisation mechanism based on ordered logic. This powerful mechanism supports both positive and negative authorisations as well as rule derivations and default propositions. Other notable features of this system include the distinction between weak and strong authorisations, support for administrative authorisation delegation and more importantly, conflict resolution.

Jajodia, et. al. [13] argued that most authorisation system models work on a single specific access control policy. Although it is theoretically possible for such systems to handle multiple policies, in practice, only one specific policy can be applied in a given system. As a solution to this problem, they proposed a general access control framework whose main feature is its flexibility to handle multiple policies in one system. Other features of this framework include support for groups and roles, conflict resolution mechanisms and support for different decision strategies.

These systems, effective as they are, lack the details necessary to address the issues involved in the implementation of such systems.

The *Policy Description Language*, or *PDL*, developed by Lobo, et. al. [18], is a language for representing event and action oriented generic policies. *PDL* is later extended by Chomicki, et. al. [8] to include *policy monitors* which, in effect, are policy constraints. Bertino, et. al. [7], again took *PDL* a step further by extending *policy monitors* to allow users to express preferred constraints. While these generic languages are expressive enough to be used for access control systems, systems built for such languages will not have the ability to dynamically update the policies.

To overcome these limitations, we propose PolicyUpdater. This access control system, with its own access control language, provides the following: (1) a formal logic-based representation of policies, with variable resolution and default propositions, (2) a mechanism to conditionally and dynamically perform a sequence of policy updates, and (3) a means of evaluating queries against the policies.

The rest of the paper is organised as follows. In Section 2, the paper introduces language $\mathcal{L}$, with its formal syntax, semantics and some examples. In Section 3, the issues

of consistency and query evaluation are addressed. The implementation, as discussed in Section 4, gives an overview of the PolicyUpdater system as a whole, with its internal and external components. The section also includes a few algorithms that outline the underlying mechanisms as well as some experimental results that show the relationship between input size and execution time. The case study presented in Section 5 describes an application of the PolicyUpdater system: an access control system for web servers. Section 6 outlines the issues involved in extending the system to include temporal authorisations. Finally, Section 7 contains a summary of the paper.

The PolicyUpdater system was originally introduced in the conference proceedings paper [9]. Another conference proceedings paper [10] focuses on a web server authorisation system based on the core PolicyUpdater system.

2 Language $\mathcal{L}$

Language $\mathcal{L}$ is a first-order logic language that represents 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. C-style comments delimited by the "/*" and "*/" characters may appear anywhere in the logic program.

2.1.1 Components of Language $\mathcal{L}$

Each language $\mathcal{L}$ statement is composed of the following components: identifiers, atoms, facts and expressions.

Identifiers. The most basic unit of language $\mathcal{L}$ is the identifier. Identifiers are used to represent the different components of the language. They are classified into three main categories:

- *Entity Identifiers* represent constant entities that make up a logical atom. They are further divided into three types, with each type again divided into the *singular entity* and *group entity* sub-types:
  - *Subjects*: e.g. alice, lecturers, user-group.
  - *Access Rights*: e.g. read, write, own.
  - *Objects*: e.g. file, database, directory.

  An entity identifier is defined as a single, lower-case alphabetic character, followed by 0 to 127 alphanumeric and underscore characters. The following regular expression shows the syntax of entity identifiers:

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

  `[a-z]([a-zA-Z0-9_]){0,127}`
- *Variable Identifiers* are used as place-holders for entity identifiers. To distinguish them from entity and policy update identifiers, variable identifiers are prefixed with an upper-case character, followed by 0 to 127 alphanumeric and underscore characters. The following regular expression shows the syntax of variable identifiers:

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

Atoms. An atom is composed of a relation with 2 to 3 entity or variable identifiers that represents a logical relationship between the entities. There are three 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. singular subject and group subject).

  `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>)`

Atoms that contain no variables, i.e. composed entirely of entity identifiers, are called *ground atoms*.

Facts. A fact states 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>
```

Note that facts may be made up of atoms that contain variable identifiers. Facts with no variable occurrences are called *ground facts*.

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

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

Expressions that are made up of only ground facts are called *ground expressions*.

2.1.2 Definition Statements

These statements are used to define the different rules that make up the policy base.

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]
  <entity_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. A constraint statement is a logical rule that holds regardless of any changes that may occur when the policy base is updated. Constraint rules are true in the initial state and remain true after any policy update.

The constraint syntax below shows that in any state of the policy base, expression $exp1$ holds if expression $exp2$ is true and there is no evidence that $exp3$ is true. The $with\ absence$ clause allows constraints to have a default proposition behaviour, where the absence of proof that an expression holds satisfies the clause condition of the proposition.

It is important to note that the expressions $exp1$, $exp2$ and $exp3$ may be non-ground expressions, which means an identifier occurring within these expressions may be a variable.

```
always <exp1>
  [implied by <exp2>
  [with absence <exp3>]];
```

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

```
<upd_id>([<var_id>[, ...]])
  causes <exp1>
  [if <exp2>];
```

upd_id is the policy update identifier to be used in referencing this policy update. The optional parameter var_id is a list which contains the variable identifiers occurring in expressions $exp1$ and $exp2$ and will be eventually replaced by entity identifiers when the update is referenced. The postcondition expression $exp1$ is an expression that will hold in the state after this update is applied. The expression $exp2$ 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.

2.1.3 Directive Statements

These statements are used to issue policy update and query directives to the PolicyUpdater system.

Policy Update Directives. The policy update sequence list contains a list of references to define 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 against which queries can be evaluated.

The following four directives are used for policy update sequence list manipulation.

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

```
seq add <upd_id>([<elt_id>[, ...]]);
```

where upd_id is the identifier of a defined policy update and the elt_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> <upd_id>([<elt_id>[, ...]])
```

where n is the ordinal index of the policy update in the sequence list starting at 0. upd_id is the policy update identifier and the elt_id list is the comma-separated list of entity identifiers used to replace the variable identifier place-holders.

– *Removing an update from the sequence.* The syntax below shows the directive used 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 does not get 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 indicate that $grp3$ does 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) &&
  holds(grp1, read, file) &&
  subst(grp2, grp1);

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(grp1, read, file);
query holds(alice, 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.

The semantics of language $\mathcal{L}$ is based on the well-known answer set (stable model) semantics of extended logic programs proposed by Gelfond and Lifschitz [11]. The definition below formally defines the answer set of a logic program.

Definition 1 Given an extended logic program π composed of ground facts and rules that do not have the negation-as-failure operator not and a set $\mathcal{F}$ of all ground facts in π. A set λ is then said to be an answer set of π if it is the smallest set that satisfies the following conditions:

1. For any rule of the form $\rho_0 \leftarrow \rho_1, \ldots, \rho_n$ where $n \geq 1$, if $\rho_1, \ldots, \rho_n \in \lambda$, then $\rho_0 \in \lambda$.
2. If λ contains a pair of complementary facts (i.e. a fact and its negation), then $\lambda = \mathcal{F}$.

For a ground extended logic program π that is composed of rules that may have the negation-as-failure operator not, a set λ is the answer set of π if and only if λ is the answer set of π', where π' is obtained from π by deleting the following:

1. Each rule that contains a fact of the form $not\ \rho$ in its body where $\rho \in \lambda$.
2. All facts of the form $not\ \rho$ in the bodies of the remaining rules.

2.2.1 Domain Description of Language $\mathcal{L}$

The definition below gives a formal definition of the domain description of language $\mathcal{L}$.

Definition 2 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 ordered 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.

2.2.2 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 below shows that a new state PB' is generated from the current state PB after the policy update u is applied.

$$PB \vec{u} PB'$$

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 function takes a policy update reference u ($u \in \psi$) and the current state σ as input arguments and returns the resulting state σ' after update u has been applied to state σ:

$$\sigma' = Res(u, \sigma)$$

Given an initial state S_0 and a policy update sequence list ψ, each state σ_i ($0 \leq i \leq |\psi|$) may be represented as follows:

$$\begin{aligned}
&\sigma_0 = S_0 \\
&\sigma_1 = Res(u_0, \sigma_0) \\
&\vdots \\
&\sigma_{|\psi|} = Res(u_{|\psi|-1}, \sigma_{|\psi|-1})
\end{aligned}$$

Substituting each state with a recursive call to the Res function, the final state $S_{|\psi|}$ is defined as follows:

$$S_{|\psi|} = Res(u_{|\psi|-1}, Res(\ldots, Res(u_0, S_0)))$$

Entities. The entity set $\mathcal{E}$ is the 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 $\mathcal{E}_{og}$. Each entity in set $\mathcal{E}$ corresponds directly to the *entity identifiers* of language $\mathcal{L}$.

$$\begin{aligned}
&\mathcal{E} = \mathcal{E}_s \cup \mathcal{E}_a \cup \mathcal{E}_o \\
&\mathcal{E}_s = \mathcal{E}_{ss} \cup \mathcal{E}_{sg} \\
&\mathcal{E}_a = \mathcal{E}_{as} \cup \mathcal{E}_{ag} \\
&\mathcal{E}_o = \mathcal{E}_{os} \cup \mathcal{E}_{og}
\end{aligned}$$

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 two to three entities, as with atoms of language $\mathcal{L}$. Furthermore, atoms of language $\mathcal{L}^*$ extends this definition by defining the state of the policy base in which the relationship holds. 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.

$$\begin{aligned}
&\mathcal{A}^\sigma = \mathcal{A}^\sigma_h \cup \mathcal{A}^\sigma_m \cup \mathcal{A}^\sigma_s \\
&\mathcal{A}^\sigma_h = \{\hat{holds}(s, a, o, \sigma) \mid s \in \mathcal{E}_s, a \in \mathcal{E}_a, o \in \mathcal{E}_o\} \\
&\mathcal{A}^\sigma_m = \mathcal{A}^\sigma_{ms} \cup \mathcal{A}^\sigma_{ma} \cup \mathcal{A}^\sigma_{mo} \\
&\mathcal{A}^\sigma_s = \mathcal{A}^\sigma_{ss} \cup \mathcal{A}^\sigma_{sa} \cup \mathcal{A}^\sigma_{so} \\
&\mathcal{A}^\sigma_{ms} = \{\hat{memb}(e, g, \sigma) \mid e \in \mathcal{E}_{ss}, g \in \mathcal{E}_{sg}\} \\
&\mathcal{A}^\sigma_{ma} = \{\hat{memb}(e, g, \sigma) \mid e \in \mathcal{E}_{as}, g \in \mathcal{E}_{ag}\} \\
&\mathcal{A}^\sigma_{mo} = \{\hat{memb}(e, g, \sigma) \mid e \in \mathcal{E}_{os}, g \in \mathcal{E}_{og}\} \\
&\mathcal{A}^\sigma_{ss} = \{\hat{subst}(g_1, g_2, \sigma) \mid g_1, g_2 \in \mathcal{E}_{sg}\} \\
&\mathcal{A}^\sigma_{sa} = \{\hat{subst}(g_1, g_2, \sigma) \mid g_1, g_2 \in \mathcal{E}_{ag}\} \\
&\mathcal{A}^\sigma_{so} = \{\hat{subst}(g_1, g_2, \sigma) \mid g_1, g_2 \in \mathcal{E}_{og}\}
\end{aligned}$$

Facts. A fact is a logical statement that makes a claim that an atom either holds or does not hold at a particular state. The following is the formal definition of fact $\hat{\rho}$ in state σ:

$$\hat{\rho}^\sigma = [\neg]\hat{\alpha}, \hat{\alpha} \in \mathcal{A}^\sigma$$

2.2.3 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 $\mathcal{D}_\mathcal{L}$ are provided by the answer sets of the extended logic 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 $\hat{\alpha}$ of language $\mathcal{L}^*$ at some state σ and another state σ'. The function returns an equivalent atom of the same type and with the same entities, but in the new state specified.

$$CopyAtom(\hat{\alpha}, \sigma') = \begin{cases} \hat{holds}(s, a, o, \sigma'), \text{ if } \hat{\alpha} = \hat{holds}(s, a, o, \sigma) \\ \hat{memb}(e, g, \sigma'), \text{ if } \hat{\alpha} = \hat{memb}(e, g, \sigma) \\ \hat{subst}(g_1, g_2, \sigma'), \text{ if } \hat{\alpha} = \hat{subst}(g_1, g_2, \sigma) \end{cases}$$

Another function, $TransAtom$, takes an atom α of language $\mathcal{L}$ and an arbitrary state σ and returns the equivalent atom of language $\mathcal{L}^*$.

$$TransAtom(\alpha, \sigma) = \begin{cases} \hat{holds}(s, a, o, \sigma), \text{ if } \alpha = holds(s, a, o) \\ \hat{memb}(e, g, \sigma), \text{ if } \alpha = memb(e, g) \\ \hat{subst}(g_1, g_2, \sigma), \text{ if } \alpha = subst(g_1, g_2) \end{cases}$$

The $TransFact$ function 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. The process of 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 $\mathcal{L}^*$. Given the following *initially* statement in language $\mathcal{L}$:

`initially` ρ_0 `&&` ... `&&` ρ_n`;`

The language $\mathcal{L}^*$ translation of this statement is shown below:

$$\begin{aligned}
&\hat{\rho}_0 \leftarrow \\
&\vdots \\
&\hat{\rho}_n \leftarrow
\end{aligned}$$

where
$\hat{\rho}_i = TransFact(\rho_i, S_0)$,
$0 \leq i \leq n$

As shown above, the number of initial fact rules generated from the translation is the number of facts n in the given language $\mathcal{L}$ initial fact expression.

The following code shows a more realistic example of language $\mathcal{L}$ *initially* statements:

```
initially
  holds(admins, read, sys_data) &&
  memb(alice, admins);

initially
  memb(bob, admins);
```

In language $\mathcal{L}^*$, the above statements are translated to:

$\hat{holds}(admins, read, sys\_data, S_0) \leftarrow$
$\hat{memb}(alice, admins, S_0) \leftarrow$
$\hat{memb}(bob, admins, S_0) \leftarrow$

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 language $\mathcal{L}$ 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.

$a_0 \leftarrow b_0, \ldots, b_n, not\ c_0, \ldots, not\ c_o$
$\vdots$
$a_m \leftarrow b_0, \ldots, b_n, not\ c_0, \ldots, not\ c_o$

Under the definition of constraint rules, each of the rules listed above must be made to hold in all states as defined by the sequence list ψ. This can be accomplished by translating each of the above rules to a set of $|\psi|$ rules, one for each state.

$\hat{a}_0^{S_0} \leftarrow \hat{b}_0^{S_0}, \ldots, \hat{b}_n^{S_0}, not\ \hat{c}_0^{S_0}, \ldots, not\ \hat{c}_o^{S_0}$
$\vdots$
$\hat{a}_0^{S_{|\psi|}} \leftarrow \hat{b}_0^{S_{|\psi|}}, \ldots, \hat{b}_n^{S_{|\psi|}}, not\ \hat{c}_0^{S_{|\psi|}}, \ldots, not\ \hat{c}_o^{S_{|\psi|}}$
$\vdots$
$\hat{a}_m^{S_0} \leftarrow \hat{b}_0^{S_0}, \ldots, \hat{b}_n^{S_0}, not\ \hat{c}_0^{S_0}, \ldots, not\ \hat{c}_o^{S_0}$
$\vdots$
$\hat{a}_m^{S_{|\psi|}} \leftarrow \hat{b}_0^{S_{|\psi|}}, \ldots, \hat{b}_n^{S_{|\psi|}}, not\ \hat{c}_0^{S_{|\psi|}}, \ldots, not\ \hat{c}_o^{S_{|\psi|}}$

where
$\hat{a}_i^\sigma = TransFact(a_i, \sigma), 0 \leq i \leq m,$
$\hat{b}_j^\sigma = TransFact(b_j, \sigma), 0 \leq j \leq n,$
$\hat{c}_k^\sigma = TransFact(c_k, \sigma), 0 \leq k \leq o,$
$S_0 \leq \sigma \leq S_{|\psi|}$

For a given language $\mathcal{L}$ constraint rule, the number of constraint rules generated in the translation is:

$m\ |\psi|$

where
m is the number of facts in the $always$ clause
$|\psi|$ is the number of states

The example below shows how the following language $\mathcal{L}$ code fragment is translated to language $\mathcal{L}^*$:

```
always
  holds(alice, read, data) &&
  holds(alice, write, data)
implied by
  memb(alice, admin)
with absence
  !holds(alice, own, data);
```

Given a policy update reference in the sequence list ψ (i.e. $|\psi| = 1$), the language $\mathcal{L}^*$ equivalent is as follows:

$\hat{holds}(alice, read, data, S_0) \leftarrow$
$\quad \hat{memb}(alice, admin, S_0),$
$\quad not\ \neg\hat{holds}(alice, own, data, S_0)$
$\hat{holds}(alice, write, data, S_0) \leftarrow$
$\quad \hat{memb}(alice, admin, S_0),$
$\quad not\ \neg\hat{holds}(alice, own, data, S_0)$

$\hat{holds}(alice, read, data, S_1) \leftarrow$
$\quad \hat{memb}(alice, admin, S_1),$
$\quad not\ \neg\hat{holds}(alice, own, data, S_1)$
$\hat{holds}(alice, write, data, S_1) \leftarrow$
$\quad \hat{memb}(alice, admin, S_1),$
$\quad not\ \neg\hat{holds}(alice, own, data, S_1)$

Policy Update Rules. Given that $m, n \geq 0$, all occurrences of variable place-holders grounded to entity identifiers, a policy update u in language $\mathcal{L}$ is in the form:

```
u causes a_0 && ... && a_m
if b_0 && ... && b_n;
```

In language $\mathcal{L}^*$, such policy updates may be represented as a set of implications, with each fact a in the postcondition expression as the consequent and precondition expression b 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.

$\hat{a}_0 \leftarrow \hat{b}_0, \ldots, \hat{b}_n$
$\vdots$
$\hat{a}_m \leftarrow \hat{b}_0, \ldots, \hat{b}_n$

where
$\hat{a}_i = TransFact(a_i, Res(u, \sigma)), 0 \leq i \leq m,$
$\hat{b}_j = TransFact(b_j, \sigma), 0 \leq j \leq n$

Intuitively, given a language $\mathcal{L}$ policy update definition, the number of language $\mathcal{L}^*$ rules generated in the translation is m, which is the number of facts in the postcondition expression.

For example, given the following 2 language $\mathcal{L}$ policy update definitions:

```
grant_read()
  causes holds(alice, read, file)
  if memb(alice, readers);

grant_write()
  causes holds(alice, write, file)
  if memb(alice, writers);
```

Given the update sequence list ψ contains $\{grant\_read, grant\_write\}$, the above statements are written in language $\mathcal{L}^*$ as:

$$\hat{holds}(alice, read, file, S_1) \leftarrow \hat{memb}(alice, readers, S_0)$$

$$\hat{holds}(alice, write, file, S_2) \leftarrow \hat{memb}(alice, writers, S_1)$$

Additional Constraints. In addition to the translations discussed above, there are a few other implicit constraint rules implied by language $\mathcal{L}$ that need to be explicitly defined in language $\mathcal{L}^*$.

– *Inheritance rules.* All properties held by a group is 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 group to propagate to its members and subsets that do not already hold the negation of the properties.

1. Subject Group Membership Inheritance
$\forall\ (s_s, s_g, a, o, \sigma),$
$\hat{holds}(s_s, a, o, \sigma) \leftarrow \hat{holds}(s_g, a, o, \sigma), \hat{memb}(s_s, s_g, \sigma),$
$not\ \neg\hat{holds}(s_s, a, o, \sigma)$
$\neg\hat{holds}(s_s, a, o, \sigma) \leftarrow \neg\hat{holds}(s_g, a, o, \sigma), \hat{memb}(s_s, s_g, \sigma)$
where
$s_s \in \mathcal{E}_{ss}, s_g \in \mathcal{E}_{sg}, a \in \mathcal{E}_a, o \in \mathcal{E}_o, S_0 \leq \sigma \leq S_{|\psi|}$
2. Access Right Group Membership Inheritance
$\forall\ (s, a_s, a_g, o, \sigma),$
$\hat{holds}(s, a_s, o, \sigma) \leftarrow \hat{holds}(s, a_g, o, \sigma), \hat{memb}(a_s, a_g, \sigma),$
$not\ \neg\hat{holds}(s, a_s, o, \sigma)$
$\neg\hat{holds}(s, a_s, o, \sigma) \leftarrow \neg\hat{holds}(s, a_g, o, \sigma), \hat{memb}(a_s, a_g, \sigma)$
where
$s \in \mathcal{E}_s, a_s \in \mathcal{E}_{as}, a_g \in \mathcal{E}_{ag}, o \in \mathcal{E}_o, S_0 \leq \sigma \leq S_{|\psi|}$
3. Object Group Membership Inheritance
$\forall\ (s, a, o_s, o_g, \sigma),$
$\hat{holds}(s, a, o_s, \sigma) \leftarrow \hat{holds}(s, a, o_g, \sigma), \hat{memb}(o_s, o_g, \sigma),$
$not\ \neg\hat{holds}(s, a, o_s, \sigma)$
$\neg\hat{holds}(s, a, o_s, \sigma) \leftarrow \neg\hat{holds}(s, a, o_g, \sigma), \hat{memb}(o_s, o_g, \sigma)$
where
$s \in \mathcal{E}_s, a \in \mathcal{E}_a, o_s \in \mathcal{E}_{os}, o_g \in \mathcal{E}_{og}, S_0 \leq \sigma \leq S_{|\psi|}$
4. Subject Group Subset Inheritance
$\forall\ (s_{g1}, s_{g2}, a, o, \sigma),$
$\hat{holds}(s_{g1}, a, o, \sigma) \leftarrow \hat{holds}(s_{g2}, a, o, \sigma), \hat{subst}(s_{g1}, s_{g2}, \sigma),$
$not\ \neg\hat{holds}(s_{g1}, a, o, \sigma)$
$\neg\hat{holds}(s_{g1}, a, o, \sigma) \leftarrow \neg\hat{holds}(s_{g2}, a, o, \sigma), \hat{subst}(s_{g1}, s_{g2}, \sigma)$
where
$s_{g1}, s_{g2} \in \mathcal{E}_{sg}, a \in \mathcal{E}_a, o \in \mathcal{E}_o, s_{g1} \neq s_{g2},$
$S_0 \leq \sigma \leq S_{|\psi|}$
5. Access Right Group Subset Inheritance
$\forall\ (s, a_{g1}, a_{g2}, o, \sigma),$
$\hat{holds}(s, a_{g1}, o, \sigma) \leftarrow \hat{holds}(s, a_{g2}, o, \sigma), \hat{subst}(a_{g1}, a_{g2}, \sigma),$
$not\ \neg\ \hat{holds}(s, a_{g1}, o, \sigma)$
$\neg\hat{holds}(s, a_{g1}, o, \sigma) \leftarrow \neg\hat{holds}(s, a_{g2}, o, \sigma), \hat{subst}(a_{g1}, a_{g2}, \sigma)$
where
$s \in \mathcal{E}_s, a_{g1}, a_{g2} \in \mathcal{E}_{ag}, o \in \mathcal{E}_o, a_{g1} \neq a_{g2},$
$S_0 \leq \sigma \leq S_{|\psi|}$
6. Object Group Subset Inheritance
$\forall\ (s, a, o_{g1}, o_{g2}, \sigma),$
$\hat{holds}(s, a, o_{g1}, \sigma) \leftarrow \hat{holds}(s, a, o_{g2}, \sigma), \hat{subst}(o_{g1}, o_{g2}, \sigma),$
$not\ \neg\hat{holds}(s, a, o_{g1}, \sigma)$
$\neg\hat{holds}(s, a, o_{g1}, \sigma) \leftarrow \neg\hat{holds}(s, a, o_{g2}, \sigma), \hat{subst}(o_{g1}, o_{g2}, \sigma)$

where
$s \in \mathcal{E}_s$, $a \in \mathcal{E}_a$, o_{g1}, $o_{g2} \in \mathcal{E}_{og}$, $o_{g1} \neq o_{g2}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

– *Transitivity rules.* Given three distinct groups 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 of subject, access right and object groups hold:

1. Subject Group Transitivity
$\forall\ (sg_1, sg_2, sg_3, \sigma)$,
$\hat{subst}(sg_1, sg_3, \sigma) \leftarrow$
$\hat{subst}(sg_1, sg_2, \sigma), \hat{subst}(sg_2, sg_3, \sigma)$
where
$sg_1, sg_2, sg_3 \in \mathcal{E}_{sg}$, $sg_1 \neq sg_2 \neq sg_3$,
$S_0 \leq \sigma \leq S_{|\psi|}$
2. Access Right Group Transitivity
$\forall\ (ag_1, ag_2, ag_3, \sigma)$,
$\hat{subst}(ag_1, ag_3, \sigma) \leftarrow$
$\hat{subst}(ag_1, ag_2, \sigma), \hat{subst}(ag_2, ag_3, \sigma)$
where
$ag_1, ag_2, ag_3 \in \mathcal{E}_{ag}$, $ag_1 \neq ag_2 \neq ag_3$,
$S_0 \leq \sigma \leq S_{|\psi|}$
3. Object Group Transitivity
$\forall\ (og_1, og_2, og_3, \sigma)$,
$\hat{subst}(og_1, og_3, \sigma) \leftarrow$
$\hat{subst}(og_1, og_2, \sigma), \hat{subst}(og_2, og_3, \sigma)$
where
$og_1, og_2, og_3 \in \mathcal{E}_{og}$, $og_1 \neq og_2 \neq og_3$,
$S_0 \leq \sigma \leq S_{|\psi|}$

– *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:
$\forall\ (\hat{\alpha}, u)\ \exists \hat{\alpha}'$,
$\hat{\alpha}' \leftarrow \hat{\alpha}, not\ \neg\ \hat{\alpha}'$
$\neg\ \hat{\alpha}' \leftarrow \neg\ \hat{\alpha}, not\ \hat{\alpha}'$
where
$\hat{\alpha} \in \mathcal{A}^\sigma$, $u \in \psi$, $\hat{\alpha}' = CopyAtom(\hat{\alpha}, Res(u, \sigma))$

– *Identity rules.* Finally, explicit rules must be given to show that every set is a subset of itself.
$\forall\ (g, \sigma)$,
$\hat{subst}(g, g, \sigma)$
where
$g \in (\mathcal{E}_{sg} \cup \mathcal{E}_{ag} \cup \mathcal{E}_{og})$, $S_0 \leq \sigma \leq S_{|\psi|}$

Definition 3 Given a domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$, the 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, (6) inertial rules, and (7) identity rules as described above.

The domain description $\mathcal{D}_\mathcal{L}$ of language $\mathcal{L}$ is said to be *consistent* if and only if the translation $Trans(\mathcal{D}_\mathcal{L})$ has a consistent answer set.

Appendix A shows the language $\mathcal{L}^*$ translation of the language $\mathcal{L}$ code listing shown in Example 1. Note that given a domain description $\mathcal{D}_\mathcal{L}$, the translation $Trans(\mathcal{D}_\mathcal{L})$ may contain more rules than the original statements in $\mathcal{D}_\mathcal{L}$. However, as the theorem below defines the maximum number of rules generated in a translation $Trans(\mathcal{D}_\mathcal{L})$, it shows that the size of a translated domain $|Trans(\mathcal{D}_\mathcal{L})|$ can only be polynomially larger than the size of the given domain $|\mathcal{D}_\mathcal{L}|$. Therefore, from a computational viewpoint, computing the answer sets of $Trans(\mathcal{D}_\mathcal{L})$ is always feasible.

Theorem 1 (Translation Size) *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 $\mathcal{E}$ containing all the entities 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 the expression of any $initially$ statement in $\mathcal{S}_i$; the maximum number of facts M_c in the $always$ clause expression of any constraint statement in $\mathcal{S}_c$; the maximum number of facts M_u in the postcondition expression of any policy update statement in $\mathcal{S}_u$; and finally the policy update sequence list ψ, then the maximum size of the translation $Trans(\mathcal{D}_\mathcal{L})$ is:*

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

Proof From Definition 3, it follows that the size of a language $\mathcal{L}^*$ translation is:

$$
\begin{aligned}
&|Trans(\mathcal{D}_\mathcal{L})| = \\
&\quad |\mathcal{F}_{in}| + |\mathcal{F}_{co}| + |\mathcal{F}_{up}| + |\mathcal{F}_{ih}| + |\mathcal{F}_{tr}| + |\mathcal{F}_{ie}| + |\mathcal{F}_{id}|
\end{aligned}
$$

where $\mathcal{F}_{in}$, $\mathcal{F}_{co}$, $\mathcal{F}_{up}$, $\mathcal{F}_{ih}$, $\mathcal{F}_{tr}$, $\mathcal{F}_{ie}$, and $\mathcal{F}_{id}$ are the sets of initial fact rules, constraint rules, policy update rules, inheritance rules, transitivity rules, inertial rules, and identity rules, respectively.

As no $initially$ statement in $\mathcal{S}_i$ contain an expression with more than M_i facts, the maximum number of initial fact rules generated in the translation is:

$$|\mathcal{F}_{in}| \leq M_i\ |\mathcal{S}_i|$$

Each language $\mathcal{L}$ constraint statement in $\mathcal{S}_c$ corresponds to n rules in language $\mathcal{L}^*$, where n is the number of policy update states times the number of facts in the *always* clause of the statement. With M_c as the maximal number of facts in the *always* clause of any constraint statement, we have:

$|\mathcal{F}_{co}| \leq |\psi|\ M_c\ |\mathcal{S}_c|$

For policy update statements, only those that are applied are actually translated to language $\mathcal{L}^*$. With M_u as the maximal number of facts in the postcondition expression of any applied policy update statement, we have:

$|\mathcal{F}_{up}| \leq |\psi|\ M_u$

The total number of inheritance rules generated in the translation is the sum of the number of member inheritance rules and the number of subset inheritance rules:

$|\mathcal{F}_{ih}| = |\mathcal{F}_{ih_m}| + |\mathcal{F}_{ih_s}|$

Since the membership inheritance rules show the relationships between every possible combination of single and group entities times the number of states times 2 (for negative facts), we have:

$$\begin{aligned}&|\mathcal{F}_{ih_m}| = \\ &\quad 2\ |\psi|\ |\mathcal{E}_{ss}|\ |\mathcal{E}_{sg}|\ |\mathcal{E}_a|\ |\mathcal{E}_o| + \\ &\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_{as}|\ |\mathcal{E}_{ag}|\ |\mathcal{E}_o| + \\ &\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_a|\ |\mathcal{E}_{os}|\ |\mathcal{E}_{og}|\end{aligned}$$

For subset inheritance rules, only the relationships between group entities are considered:

$$\begin{aligned}&|\mathcal{F}_{ih_s}| = \\ &\quad 2\ |\psi|\ |\mathcal{E}_{sg}|^2\ |\mathcal{E}_a|\ |\mathcal{E}_o| + \\ &\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_{ag}|^2\ |\mathcal{E}_o| + \\ &\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_a|\ |\mathcal{E}_{og}|^2\end{aligned}$$

As transitivity rules enumerate every possible combinations of any three group entities, for each entity type, the total number of transitivity rules is shown below:

$|\mathcal{F}_{tr}| = |\psi|\ (|\mathcal{E}_{sg}|^3 + |\mathcal{E}_{ag}|^3 + |\mathcal{E}_{og}|^3)$

A single atom in language $\mathcal{L}$ corresponds to n inertial rules in language $\mathcal{L}^*$, where n is the number of states times 2 (for negative facts). This means the total number of inertial rules generated is:

$|\mathcal{F}_{ie}| = 2\ |\psi|\ |\mathcal{A}|$

Lastly, the total number of identity rules is equal to the total number of group entities times the number of states:

$|\mathcal{F}_{id}| = |\psi|\ (|\mathcal{E}_{sg}| + |\mathcal{E}_{ag}| + |\mathcal{E}_{og}|)$

□

3 Domain Consistency Checking and Evaluation

A domain description of language $\mathcal{L}$ must be consistent in order to 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 the above issues can be considered, a few notational constructs should 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
```

Let γ_{int} be an initial fact definition statement, γ_{con} a constraint definition statement, and γ_{upd} a policy update definition statement, where $\gamma_{int}, \gamma_{con}, \gamma_{upd} \in \mathcal{D}_\mathcal{L}$. We then define the following set constructor functions:

$$\begin{aligned}&\mathcal{F}^+_{int}(\gamma_{int}) = \{a_z \mid 0 \leq z \leq m\} \\ &\mathcal{F}^-_{int}(\gamma_{int}) = \{b_z \mid 0 \leq z \leq n\} \\ &\mathcal{F}^+_{con}(\gamma_{upd}) = \{c_z \mid 0 \leq z \leq o\} \\ &\mathcal{F}^-_{con}(\gamma_{upd}) = \{d_z \mid 0 \leq z \leq p\} \\ &\mathcal{F}^+_{upd}(\gamma_{con}) = \{i_z \mid 0 \leq z \leq u\} \\ &\mathcal{F}^-_{upd}(\gamma_{con}) = \{j_z \mid 0 \leq z \leq v\}\end{aligned}$$

Using these functions, we define the following sets of ground facts:

$$\begin{aligned}&\mathcal{F}^+_{int} = \{\rho \mid \rho \in \mathcal{F}^+_{int}(\gamma_{int}), \gamma_{int} \in \mathcal{D}_\mathcal{L}\} \\ &\mathcal{F}^-_{int} = \{\rho \mid \rho \in \mathcal{F}^-_{int}(\gamma_{int}), \gamma_{int} \in \mathcal{D}_\mathcal{L}\} \\ &\mathcal{F}^+_{con} = \{\rho \mid \rho \in \mathcal{F}^+_{con}(\gamma_{con}), \gamma_{con} \in \mathcal{D}_\mathcal{L}\} \\ &\mathcal{F}^-_{con} = \{\rho \mid \rho \in \mathcal{F}^-_{con}(\gamma_{con}), \gamma_{con} \in \mathcal{D}_\mathcal{L}\} \\ &\mathcal{F}^+_{upd} = \{\rho \mid \rho \in \mathcal{F}^+_{upd}(\gamma_{upd}), \gamma_{upd} \in \mathcal{D}_\mathcal{L}\} \\ &\mathcal{F}^-_{upd} = \{\rho \mid \rho \in \mathcal{F}^-_{upd}(\gamma_{upd}), \gamma_{upd} \in \mathcal{D}_\mathcal{L}\}\end{aligned}$$

Additionally, we use the complementary set notation $\overline{\mathcal{F}}$ to denote a set containing the negation of facts in set $\mathcal{F}$.

$\overline{\mathcal{F}} = \{\neg\rho \mid \rho \in \mathcal{F}\}$.

Let γ be an initial, constraint or policy update definition statement of language $\mathcal{L}$. We then define the following functions:

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

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

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

Definition 4 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 5 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 6 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 any two constraint statements γ and γ' in $\mathcal{D}_\mathcal{L}$, including $\gamma = \gamma'$, $Def(\gamma) \cap Eff(\gamma') = \emptyset$.
3. For all constraint statements γ in $\mathcal{D}_\mathcal{L}$, $\overline{Eff(\gamma)} \cap Pre(\gamma) = \emptyset$.
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) *A* normal *domain description of language $\mathcal{L}$ is also* consistent.

Proof From Definition 3, given a normal domain description $\mathcal{D}_\mathcal{L}$, we only need to show that $Trans(\mathcal{D}_\mathcal{L})$ has at least one consistent answer set to prove that $\mathcal{D}_\mathcal{L}$ is also consistent.

Given a normal domain description $\mathcal{D}_\mathcal{L}$, Condition 2 in Definition 6 ensures that the translation $Trans(\mathcal{D}_\mathcal{L})$ do not contain rules of the following form:

$\hat{\rho}_0 \leftarrow \ldots, not\ \hat{\rho}_k, \ldots$
$\hat{\rho}_1 \leftarrow \ldots, \hat{\rho}_0, \ldots$
$\vdots$
$\hat{\rho}_{k-1} \leftarrow \ldots, \hat{\rho}_{k-2}, \ldots$
$\hat{\rho}_k \leftarrow \ldots, \hat{\rho}_{k-1}, \ldots$

The absence of these rules means $Trans(\mathcal{D}_\mathcal{L})$ is a program without negative cycles [17]. As no other rule in $\mathcal{D}_\mathcal{L}$ can cause $Trans(\mathcal{D}_\mathcal{L})$ to have these rules, we conclude that a normal domain description $\mathcal{D}_\mathcal{L}$, as defined by Definition 6, will generate an extended logic program $Trans(\mathcal{D}_\mathcal{L})$ without negative cycles. Also, from [5,17], we further conclude that the translated program $Trans(\mathcal{D}_\mathcal{L})$ must have an answer set.

Condition 1 of Definition 6 prevents rules of the following form from occurring in $Trans(\mathcal{D}_\mathcal{L})$:

$\hat{\rho}^{S_0} \leftarrow$
$\neg\hat{\rho}^{S_0} \leftarrow$

This shows that a subset of the answer set which contains facts from the initial state S_0 is consistent.

Condition 3 of Definition 6 guarantees that rules of the following form do not occur in $Trans(\mathcal{D}_\mathcal{L})$:

$\hat{\rho} \leftarrow \ldots, \neg\hat{\rho}, \ldots$

This ensures that all constraint rules translated from $\mathcal{D}_\mathcal{L}$ are consistent.

Finally, Condition 4 of Definition 6 ensures that rules in $Trans(\mathcal{D}_\mathcal{L})$ of the following form:

$\hat{\rho} \leftarrow \ldots, \hat{\rho}', \ldots$
$\neg\hat{\rho} \leftarrow \ldots, \hat{\rho}'', \ldots$

cannot both affect the answer set as the premises ρ' and ρ'' are mutually exclusive and therefore only one is true in any given state.

These guarantee that the answer set do not contain complementary facts, and therefore guarantee that the answer set is consistent.

□

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

Definition 7 Given a *consistent* domain description $\mathcal{D}_\mathcal{L}$, a ground query expression ϕ and a finite sequence list ψ, we say *query ϕ holds in $\mathcal{D}_\mathcal{L}$ after the policy updates in sequence list ψ have been applied*, denoted as

$$\mathcal{D}_\mathcal{L} \models \{\phi, \psi\}$$

if and only if

$$\forall\ (\rho, \lambda), \hat{\rho} \in \lambda$$

where
$\rho \in \phi$, $\lambda \in \Lambda$,
$\hat{\rho} = TransFact(\rho, S_{|\psi|})$,
Λ = answer sets of $Trans(\mathcal{D}_\mathcal{L})$

Definition 7 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.

Example 2 Given the language $\mathcal{L}$ code listing in Example 1 and its semantic translation in Appendix A, where the update sequence list $\psi = \{delete\_read(grp1, file)\}$. The following shows the evaluated results of each query ϕ:

$\phi_0 = holds(grp1, write, file) : TRUE$
$\phi_1 = holds(grp1, read, file) : FALSE$
$\phi_2 = holds(alice, write, file) : TRUE$
$\phi_3 = holds(alice, read, file) : FALSE$

4 Implementation

As mentioned earlier, *PolicyUpdater* is a fully-implemented system. In this section, we describe the implementation details of this system. Further technical information and source code can be found in the project homepage at:

`http://www.cit.uws.edu.au/~jcrescin/projects/PolicyUpdater`

4.1 System Structure

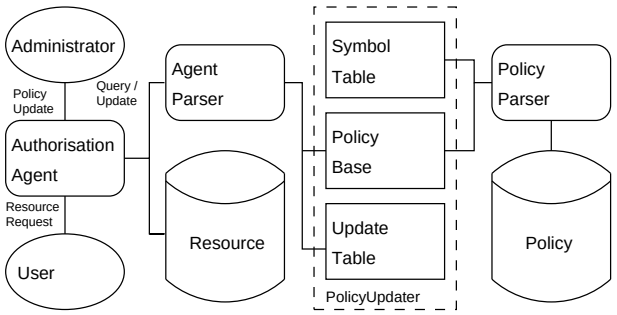

Fig. 1 Structure of PolicyUpdater

As shown in Figure 1, the PolicyUpdater system works with an authorisation agent program that queries the policy base to determine whether to allow users access to resources. Through an authorisation agent program, the PolicyUpdater system also allows administrators to dynamically update the policy base by adding or removing update directives in the policy update table.

4.1.1 Parsers

As the policy itself is written in language $\mathcal{L}$, the system uses two parsers to act as interfaces to the authorisation agent and the language $\mathcal{L}$ policy.

Policy Parser. The policy parser is responsible for correctly reading the policy file into the core PolicyUpdater system. The parser ensures that the policy file strictly adheres to the language $\mathcal{L}$ syntax then systematically stores entity identifiers into the symbol table while initial state facts, constraint expressions and policy update definitions are stored into their respective tables in the policy base.

Agent Parser. The agent parser is the direct link between the core PolicyUpdater system and the authorisation agent program. The parser's sole purpose is to receive language $\mathcal{L}$ directives from an agent, perform the directive upon the policy base and return a reply if the directive requires one. Such directives may be to query the policy base or to manipulate the policy update sequence table.

4.1.2 Data Structures

As language $\mathcal{L}$ program is parsed, each statement containing entity declarations, initial facts, constraint rules and policy updates must first be stored into a structure before the translation process is started. As shown in Appendix B, the structure is composed of the symbol table, the policy base and the policy update sequence table.

The symbol table is used to store all entity identifiers defined in the policy, while the rest of the policy definitions are stored into the policy base. On the other hand, the sequence of policy update directives are stored separately into the update table.

4.2 System Processes

The processes presented in this section shows how the language $\mathcal{L}$ policy stored in the data structures is translated into a normal logic program and how it can be dynamically updated and manipulated to evaluate queries. The flowchart in Figure 2 gives an overview of the system processes.

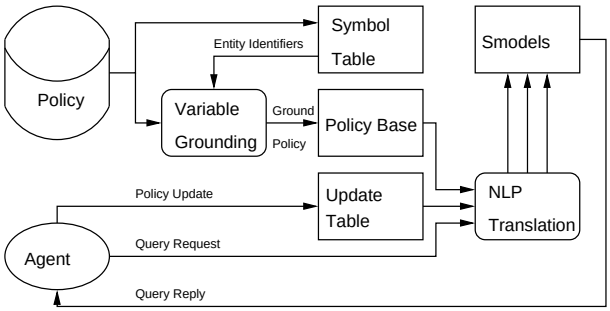

Fig. 2 System Flowchart

4.2.1 Grounding Constraint Variables

As the constraints are in the process of being added into the constraints table, each variable identifier that occurs in a constraint is grounded by replacing that constraint with a

set of constraints wherein each instance of the variable is replaced by all entity identifiers defined in the symbol table. Note that only those entity identifiers that are valid for each fact in the current constraint are used to replace the variable (e.g. only singular subject entity identifiers are used to replace an element variable occurring in a subject member fact).

For example, given that the symbol table contains three singular subject entity identifiers: $alice$, bob and $charlie$, and the following constraint:

```
always holds(SSUB, write, file)
  implied by
    holds(SSUB, read, file) &&
    memb(SSUB, students)
  with absence
    !holds(SSUB, write, file);
```

Grounding the constraint statement above yields three new constraint rules, each replacing occurrences of the variable $SSUB$ with $alice$, bob and $charlie$, respectively.

4.2.2 Policy Updates

In Section 2.2, it is shown that policy updates are performed by treating each update as a constraint. This constraint is composed of a premise, which is the precondition in the current state and a consequent, which is the postcondition of the resulting state after the application of the policy update. The resulting state in this procedure represents the updated policy.

The most crucial step in performing a policy update is the translation of the policy updates into normal logic program constraints. This step involves identifying which policy updates are to be applied from the update sequence table and then composing the required constraint from the update definition in the policy base. Once the policy update constraints are composed, they are then treated as any other constraint rules and are translated with the rest of the policy into a normal logic program.

4.2.3 Translation to Normal Logic Program

The semantics of language $\mathcal{L}$ shows that any consistent language $\mathcal{L}$ program can be translated into an equivalent extended logic program then translated again into an equivalent normal logic program. However, the implementation of such translations can be greatly simplified by translating language $\mathcal{L}$ programs directly into normal logic programs.

Removing Classical Negation. In order to remove classical negation from facts of language $\mathcal{L}$, each classically negated fact $\neg\rho$ is replaced by a new and unique positive fact ρ' that represents the negation of fact ρ. To preserve the consistency of the policy base for all facts ρ in the domain, the following constraint rule must be added:

$$FALSE \leftarrow \rho, \rho'$$

The removal process involves adding a boolean parameter to each fact to indicate whether the fact is classically negated or not. For example, given the fact:

$$\neg\, holds(alice, exec, file)$$

To remove classical negation, it is replaced by:

$$holds(alice, exec, file, false)$$

For consistency, the following constraint is added:

$$FALSE \leftarrow holds(alice, exec, file, true), holds(alice, exec, file, false)$$

Representing Facts in Propositional Form. A fact expressed in normal logic program form is composed of the atom relation, the state in which it holds and a boolean flag to indicate classical negation. For notational simplicity, this tuple may be represented by a unique positive integer i, where $0 \leq i < |\mathcal{F}|$ ($|\mathcal{F}|$ is the total number of facts in the domain). The process of translating facts of language $\mathcal{L}$ into normal logic program form is summarised by the following function:

$$i = Encode(\alpha, \sigma, \tau)$$

As shown above, the $Encode$ function takes a language $\mathcal{L}$ atom α, the state σ in which α holds, and a boolean value τ to indicate whether or not α is classically negated. $Encode$ returns a unique index i for that fact. The steps below outlines how the $Encode$ function computes the index i.

- *Enumerate all possible atoms.* By using all the entities in the symbol table, all possible language $\mathcal{L}$ atoms may be enumerated by grouping together 2 to 3 entities together. All possible atoms of type $holds$ are generated by enumerating all possible combinations of subject, access right and object entities. The set of $member$ atoms is generated from all the different combinations of singular and group entities of types subject, access right and object. Similarly, the set of $subset$ atoms is derived from different subject, access right and object group pair combinations.
- *Arrange the atoms in a predefined order.* This procedure relies on the assumption that the list of all possible atoms derived from the step above is arranged in a predefined order. In this step we ensure that the atoms are enumerated in the following order: $holds$, $subject\ member$, $access\ right\ member$, $object\ member$, $subject\ subset$, $access\ right\ subset$ and $object\ subset$. In addition to the ordering of atom types, atoms of each type are themselves sorted according to the order in which their entities appear in the symbol table.
- *Assign an ordinal index for each enumerated atom.* Since the enumerated list of atoms are ordered, consecutive positive integers may be assigned to each atom as an ordinal index i, where $0 \leq i < n$ (n is the total number of atoms enumerated).

– *Extend indexing procedure to represent facts.* At the implementation level, facts are just atoms with truth values. As such, we can treat each atom as positive facts. Since negative facts are just mirror images of their positive counterparts, their indices are calculated by adding n to the indices of the corresponding positive facts. Thus, indices i, where $n \leq i < 2n$ are negative facts while indices i, where $0 \leq i < n$ are positive facts. Furthermore, this procedure is again extended to represent the states of the facts. The process is similar: indices i, where $0 \leq i < 2n$ represent facts of state S_0, indices i, where $2n \leq i < 4n$ represent facts of state S_1, and so on.

Generating the Normal Logic Program from the Policy Base. With the language $\mathcal{L}$ policy elements stored into the storage structures (see Appendix B), a normal logic program can then be generated for evaluation. The following algorithm generates a normal logic program, given the Symbol Table Ts, Initial State Facts Table Ti, Constraint Rules Table Tc, Policy Update Definition Table Tu, and Policy Update Sequence Table Tq:

```
FUNCTION GenNLP(Ts, Ti, Tc, Tu, Tq)
  TransInitStateRules(Ti)
  TransConstRules(Tc, Tq)
  TransUpdateRules(Tu, Tq)
  GenInherRules(Ts, Tq)
  GenTransRules(Ts, Tq)
  GenInertRules(Ts, Tq)
  GenIdentRules(Ts, Tq)
  GenConsiRules(Ts, Tq)
ENDFUNCTION
```

The first three $Trans*()$ functions above perform a direct translation of language $\mathcal{L}$ statements to normal logic program. The remaining five $Gen*()$ functions generate additional constraint rules. In the following algorithms, we use the following rule constructor functions to generate normal logic program rules:

– $RuleBegin()$ marks the beginning of a new rule.
– $RuleHead(\alpha, \tau)$ generates the consequent of the rule. α is a numeric representation of an atom (e.g. returned by the $Encode()$ function) and τ is either T or F, indicating whether the atom is positive or negative (negation-as-failure).
– $RuleBody(\alpha, \tau)$ generates the premise of the rule. The parameters of this function is the same as that of the function $RuleHead()$.
– $RuleEnd()$ marks the end of a rule.

The algorithm below illustrates how initial state rules are generated from the storage structures. The process itself is straightforward: each fact in the initial state facts table is translated by the $Encode()$ function and is made the head of a new rule whose body is the literal $True$ fact.

```
FUNCTION TransInitStateRules(Ti)
  FOR i = 0 TO Len(Ti) DO
    a = Encode(Ti[i].atm, 0, Ti[i].tr)
    RuleBegin()
    RuleHead(a, T)
    RuleBody(T, T)
    RuleEnd()
  ENDDO
ENDFUNCTION
```

The constraint rules generating algorithm below works by creating a new rule that is composed of facts from the constraints table translated by the $Encode()$ function. The outer loop ensures that a rule is generated for every policy update state.

```
FUNCTION TransConstRules(Tc, Tq)
  FOR i = 0 TO Len(Tq) DO
    FOR j = 0 TO Len(Tc) DO
      RuleBegin()
      FOR k = 0 TO Len(Tc[j].exp) DO
        a = Encode(Tc[j].exp[k].atm,
                   i,
                   Tc[j].exp[k].tr)
        RuleHead(a)
      ENDDO
      FOR k = 0 TO Len(Tc[j].pcond) DO
        a = Encode(Tc[j].pcond[k].atm,
                   i,
                   Tc[j].pcond[k].tr)
        RuleHead(a, T)
      ENDDO
      FOR k = 0 TO Len(Tc[j].ncond) DO
        a = Encode(Tc[j].ncond[k].atm,
                   i,
                   Tc[j].ncond[k].tr)
        RuleHead(a, F)
      ENDDO
      RuleEnd()
    ENDDO
  ENDDO
ENDFUNCTION
```

The algorithm below generates the policy update rules from the given policy update definition table. Note that only those policy updates that also appear in the policy update sequence list are actually translated. The actual translation process is similar to that of constraint rules, except each variable that may occur within the expressions is first grounded and the policy update state of each fact in the rule head is one more than that of each fact in the rule body.

```
FUNCTION TransUpdateRules(Tu, Tq)
  FOR i = 0 TO Len(Tq) DO
    FOR j = 0 TO Len(Tu) DO
      IF Tq[i].name == Tu[j].name THEN
        e =
          GndUpdate(Tu[j], Tq[i].ilist)
        RuleBegin()
        FOR k = 0 TO Len(e.post) DO
          a = Encode(e.post[k].atm,
```

```
                    i + 1,
                    e.post[k].tr)
          RuleHead(a, T)
        ENDDO
        FOR k = 0 TO Len(e.pre) DO
          a = Encode(e.pre[k].atm,
                    i,
                    e.pre[k].tr)
          RuleBody(a, T)
        ENDDO
        RuleEnd()
      ENDIF
    ENDDO
  ENDDO
ENDFUNCTION
```

The function $GndUpdate(U, IL)$ used in the algorithm above returns a structure composed of two expressions pre and $post$, which corresponds with the pre and $post$ fields of the given policy update definition U. All variables occurring in the facts of these expressions are replaced with the corresponding entities from the given entity identifier list IL.

The function shown below generates 6 types of inheritance rules: subset subject, subset access right, subset object, membership subject, membership access right and membership object. Each of these 6 algorithms work in a similar way: a rule is generated by composing every possible combination of either subject, access right and object entities to form either a subset or membership fact. As with the contraint rule generating algorithm, each new rule generated is replicated for each policy update state.

```
FUNCTION GenInherRules(Ts, Tq)
  GenSubSubstInherRules(Ts, Tq)
  GenAccSubstInherRules(Ts, Tq)
  GenObjSubstInherRules(Ts, Tq)
  GenSubMembInherRules(Ts, Tq)
  GenAccMembInherRules(Ts, Tq)
  GenObjMembInherRules(Ts, Tq)
ENDFUNCTION
```

The function below generates all the transitivity rules. Each subject, access right and object transitivity rule generation algorithm follows a similar procedure: every possible combination of subject, access right or object group entities are used to form subset facts, then each of these facts are used to form a transitivity rule. As with inheritance rules, each transitivity rule is replicated for each policy update state.

```
FUNCTION GenTransRules(Ts, Tq)
  GenSubTransRules(Ts, Tq)
  GenAccTransRules(Ts, Tq)
  GenObjTransRules(Ts, Tq)
ENDFUNCTION
```

The inertial rules generation function below is composed of 3 functions that generate inertial rules for each atom type: holds, membership and subset. Each type of rule is generated by composing different combinations of entity identifiers together to form a fact. Each rule is then formed by stating that for each policy update state, a fact holds in the current state if it also holds in the previous state and its negation does not hold in the current state.

```
FUNCTION GenInertRules(Ts, Tq)
  GenHldsInertRules(Ts, Tq)
  GenMembInertRules(Ts, Tq)
  GenSubsInertRules(Ts, Tq)
ENDFUNCTION
```

The function $GenIdentRules()$ shown below generates the identity rules for each atom type: subject, access right and object. A simple procedure is followed by each of the 3 functions: for every subject, access right and object group entities, a subset rule is formed to show that a group is a subset of itself. As with the other rules, each rule generated by these functions is replicated for each policy update state.

```
FUNCTION GenIdentRules(Ts, Tq)
  GenSubIdentRules(Ts, Tq)
  GenAccIdentRules(Ts, Tq)
  GenObjIdentRules(Ts, Tq)
ENDFUNCTION
```

The last two functions below shows the algorithm to generate consistency rules for each atom type: holds, membership and subset. As these rules use a similar process to generate rules, only the holds consistency rule generation algorithm is shown. The rules that are generated ensure that only a fact or its negation, but never both, holds in the same policy update state.

```
FUNCTION GenConsiRules(Ts, Tq)
  GenHldsConsiRules(Ts, Tq)
  GenMembConsiRules(Ts, Tq)
  GenSubsConsiRules(Ts, Tq)
ENDFUNCTION

FUNCTION GenHldsConsiRules(Ts, Tq)
  FOR i = 0 TO Len(Tq) DO
    FOR j = 0 TO Len(Ts.s) DO
      FOR k = 0 TO Len(Ts.a) DO
        FOR l = 0 TO Len(Ts.o) DO
          ahlds.sub = Ts.s[j]
          ahlds.acc = Ts.a[k]
          ahlds.obj = Ts.o[l]
          RuleBegin()
          RuleHead(F, T)
          a = Encode(ahlds, i, T)
          RuleBody(a, T)
          a = Encode(ahlds, i, F)
          RuleBody(a, T)
          RuleEnd()
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

4.2.4 Query Evaluation

Once a normal logic program has been generated from the policy stored in the storage structure, a set of answer sets may then be generated by using the stable model semantics [21] with the *smodels*[1]program. Query evaluation then becomes possible by checking whether each fact of a given query expression holds in each generated answer set of the normal logic program.

If a given fact indeed holds in all the answer sets, it is then evaluated to be true. On the other hand, if the negation of a fact holds in every answer set, then it is evaluated to be false. A fact or its negation that does not hold in every answer set is neither true nor false, in which case the system concludes that the truth value of the fact is unknown. The algorithm below shows how, given a state S, a query expression Qe can be evaluated against a list of stable models SM, where each element in SM is a list of facts.

```
FUNCTION EvaluateExp(Qe, SM, S)
  result = T
  FOR i = 0 TO Len(Qe) DO
    rv = EvaluateFact(Qe[i], SM, S)
    IF rv == F THEN
      RETURN F
    ELSE IF rv == U THEN
      result = U
    ENDIF
  ENDDO
  RETURN result
ENDFUNCTION
```

The algorithm above attempts to evaluate each fact in the query expression. The function $EvaluateFact()$ shown below evaluates a single fact Qf in state S, against a list of stable models SM.

```
FUNCTION EvaluateFact(Qf, SM, S)
  a = Encode(Qf.atm, S, Qf.tr)
  IF IsFactIn(SM, a) THEN
    RETURN T
  ELSE
    a = Encode(Qf.atm, S, NOT Qf.tr)
    IF IsFactIn(a, SM) THEN
      RETURN F
    ELSE
      RETURN U
    ENDIF
  ENDIF
ENDFUNCTION
```

The function $IsFactIn()$ simply returns a boolean value to indicate whether or not the given fact index Fi (as returned by the $Encode()$ function) is present in every stable model in SM.

```
FUNCTION IsFactIn(Fi, SM)
  FOR i = 0 TO Len(SM) DO
    IF NOT IsIn(SM[i], Fi) THEN
      RETURN F
    ENDIF
  ENDDO
  RETURN T
ENDFUNCTION
```

[1] Smodels (`http://www.tcs.hut.fi/Software/smodels`)

4.3 Experimental Results

In this subsection, we investigate the effects of domain size over computation time. The following tests were conducted with the latest version of PolicyUpdater[2]running on an AMD Athlon XP 2000+ machine with 512 MB of RAM, running the Debian GNU/Linux 3.0r5 operating system with a plain Linux 2.4.30 kernel.

Table 1 shows the domain size for each test case. S_{E_s} and S_{E_g} are the numbers of singular and group entities, respectively; S_I is the number of initial state facts; S_C is the number of constraint rules; S_U is the number of policy update definitions; S_S is the number of policy updates in the sequence list; and S_Q is the number of facts to be queried.

	S_{E_s}	S_{E_g}	S_I	S_C	S_U	S_S	S_Q
1	4	3	3	1	1	1	4
2	24	23	3	1	1	1	4
3	104	3	3	1	1	1	4
4	4	103	3	1	1	1	4
5	24	23	103	1	1	1	4
6	24	23	3	101	1	1	4
7	24	23	3	1	101	1	4
8	24	23	3	1	101	101	4
9	24	23	3	1	1	1	104
10	24	23	103	1	101	101	4
11	24	23	3	101	101	101	4
12	24	23	103	101	101	101	104
13	104	103	103	101	101	101	104

Table 1 Thirteen test cases with different domain sizes

The language L code listing in Example 1 is used in the first test case. In the second test case, the same code is used with 20 new singular entities and 20 new group entities. Test cases 3 and 4 are similar to test case 1, except 100 new singular and group entities were added, respectively. Test cases 5 and 6 are similar to test case 2, except 100 new initial state facts and constraint rules were added, respectively. In test case 7, 100 new policy update definitions were added, and in test case 8, these policy update defintions were applied. Test case 9 is similar to test case 2, but this one tries to evaluate 100 additional query facts. Test case 11 is a combination of test cases 6 and 8. Test case 12 is a combination of test cases 5, 9 and 11. Finally, test case 13 is a combination of test cases 3 to 9, where the number of each domain component is over 100.

[2] At the time of writing, the latest version of PolicyUpdater is vlad 1.0.4.

Table 2 shows the execution times of each test case. T_C is the total time (in seconds) spent by the system to translate the language L statements to a normal logic program and to generate the answer sets. T_Q is the total time (in seconds) used by the system to evaluate all the queries. To increase result accuracy, each test was conducted 10 times. The figures in Table 2 are the averages.

	T_C	T_Q
1	0.000794	0.000472
2	0.261828	0.600932
3	0.072069	0.157254
4	14.017335	32.109291
5	0.309517	0.698068
6	0.306517	0.694729
7	0.304570	0.696636
8	15.315347	32.111353
9	0.301429	25.147113
10	15.375953	32.537575
11	15.889154	33.246048
12	15.715761	575.237985
13	?	?

Table 2 Average computation times in seconds for each test case

As shown in Table 2, the first two execution times are minimal when the domain size is small. Test 3 shows that having a large number of singular entities have a measurable, but insignificant effect on computation time. However, test 4 shows that an increase in the number of group entities have a great impact on computation speed. This is to be expected, as Section 2.2 shows that the number of group entities directly affect the number of transitivity, inheritance and identity rules generated in the translation.

Comparing test 2 with tests 5 and 6, where the number of initial state facts and constraint rules are increased by 100, respectively, we observe that that there is a slight increase in the times required to perform the computation and query evaluation. One would expect that an increase in the number of constraint rules will have more impact in execution times than an increase in initial state facts. However, in test 6, the computation times were low because only one policy update was actually applied.

Test 7 shows that increasing the number of policy update definitions has little impact on the computation times. However, as test 8 shows, if these policy updates are actually applied to the policy base, computation time increases dramatically.

Test case 9 shows that evaluating 100 additional queries has little effect on translation and computation time, but obviously affects evaluation time.

Test case 10 shows the combined effects of an increased number of policy updates and initial state facts. As expected, the times are only slightly larger than the times in test case 8, where only the number of policy updates were increased. This is due to the fact that initial state facts are translated directly into normal logic program rules. On the other hand, test case 11 shows a significant increase in both computation and evaluation times. This is expected, as the translation of a single constraint rule results in a constraint rule in every policy update state.

Test case 12 shows that although large numbers of initial state facts and query requests by themselves have little effect on performance, if combined together with the effects of a large number of policy updates, computation time is significantly increased, paticularly the query evaluation time. Note that the value of T_Q for this test is the average total time for 104 query evaluations. Using this value, each query evaluation takes an average of 5.531135 seconds to complete.

Unfortunately, the test system used in this experiment ran out of memory while performing test case 13. Again, this is expected, as the combined effects of having a large number of entities, constraint rules, policy updates and queries will result in approximately 5.7 billion rules, using the formula given in Theorem 1.

5 Case Study: Web Server Application

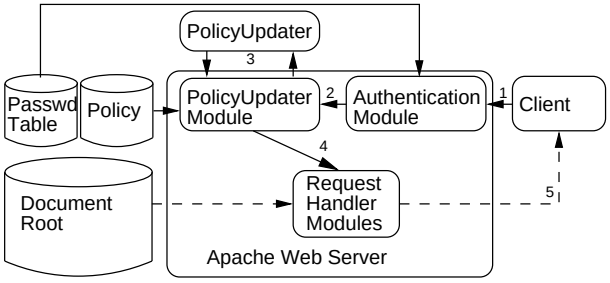

Fig. 3 PolicyUpdater module for Apache

The expressiveness of language $\mathcal{L}$ and the effectiveness of the PolicyUpdater system can be demonstrated by a web server authorisation application. In this application, the core PolicyUpdater system serves as an authorisation module for the *Apache*[3]web server.

The Apache web server provides a generic access control system as provided by its *mod_auth* and *mod_access* modules [2,15]. With this built-in access control system, Apache provides the standard HTTP *Basic* and *Digest* authentication schemes [20], as well as an authorisation system to enforce access control policies. Although the PolicyUpdater module do not provide the full functionality of Apache's built-in authorisation module *mod_auth*, it does provide a flexible logic-based authorisation mechanism.

As shown in Figure 3, Apache's Access Control module, together with its policy base, is replaced by the PolicyUpdater module and its own policy base. The sole purpose of the PolicyUpdater module is to act as an interface between the web server and the core PolicyUpdater system.

[3] Apache Web Server (`http://www.apache.org`)

The system works as follows: as the server is started, the PolicyUpdater module initialises the core PolicyUpdater system by sending the policy base. When a client makes an arbitrary HTTP request for a resource from the server (1), the client (user) is authenticated against the password table by the built-in authentication module; once the client is properly authenticated (2) the request is transferred to the PolicyUpdater module, which in turn generates a language $\mathcal{L}$ query (3) from the request details, then sends the query to the core PolicyUpdater system for evaluation; if the query is successful and access control is granted, the original request is sent to the other request handlers of the web server (4) where the request is eventually honoured; then finally (5), the resource (or acknowledgement for HTTP requests other than GET) is sent back to the client. Optionally, client can be an administrator who, after being authenticated, is presented with a special administrator interface by the module to allow the policy base to be updated.

5.1 Policy Description in Language $\mathcal{L}'$

The policy description in the policy base is written in language $\mathcal{L}'$, which is syntactically and semantically similar to language $\mathcal{L}$ except for the lack of entity identifier definitions. Entity identifiers need not be explicitly defined in the policy definition:

- *Subjects* of the authorisation policies are the users. Since all users must first be authenticated, the password table used in authentication may also be used to extract the list of subjects.
- *Access Rights* are the HTTP request methods defined by the HTTP 1.1 standard [19]: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT.
- *Objects* are the resources available in the server themselves. Assuming that the document root is a hierarchy of directories and files, each of these are mapped as a unique object of language $\mathcal{L}'$.

Like language $\mathcal{L}$, language $\mathcal{L}'$ allows the definition of initial state facts, constraint rules and policy update definitions.

5.2 Mapping the Policies to Language $\mathcal{L}$

As mentioned above, one task of the PolicyUpdater module is to generate a language $\mathcal{L}$ policy from the given language $\mathcal{L}'$ to be evaluated by the core PolicyUpdater system. This process is outlined below:

- *Generating entity identifier definitions.* Subject entities are taken from the authentication (password) table; access rights are hard-coded built-ins; and the list of objects are generated by traversing the document root for files and directories.
- *Generating additional constraints.* Additional constraint rules are generated to preserve the relationship between groups and elements. This is useful to model the assertion that unless explicitly stated, users holding particular access rights to a directory automatically hold those access rights to every file in that directory (recursively, if with subdirectories). The module makes this assertion by generating non-conditional constraint rules that state that each file (object) is a member of the directory (object group) in which it is contained.

All other language $\mathcal{L}'$ statements (initial state definitions, constraint definitions and policy update definitions) are already in language $\mathcal{L}$ form.

5.3 Evaluation of HTTP Requests

A HTTP request may be represented as a simplified tuple:

$$<usr, req\_meth, req\_res>$$

usr is the authenticated username that made the request (subject); req_meth is a standard HTTP request method (access right); and req_res is the resource associated with the request (object). Intuitively, such a tuple may be expressed as a language $\mathcal{L}$ atom:

```
holds(usr, req_meth, req_res)
```

With each request expressed as language $\mathcal{L}$ atoms, a language $\mathcal{L}$ query statement can be composed to check if the request is to be honoured:

```
query holds(usr, req_meth, req_res);
```

Once the query statement is composed, it is then sent by the PolicyUpdater module to the core PolicyUpdater system for evaluation against the policy base.

5.4 Policy Updates by Administrators

After being properly authenticated, an administrator can perform policy updates through the use of a special interface generated by the PolicyUpdater module. This interface lists all the predefined policy updates that are allowed, as defined in the policy description in language $\mathcal{L}'$, as well as all the policy updates that have been previously applied and are in effect. As with the core PolicyUpdater system, administrators are allowed only the following operations:

- Apply a policy update or a sequence of policy updates to the policy base. Note that like language $\mathcal{L}$, in language $\mathcal{L}'$ policy updates are predefined within the policy base themselves.
- Revert to a previous state of the policy base by removing a previously applied policy update from the policy base.

6 Future Research and Extension

An obvious limitation of language $\mathcal{L}$, and therefore of the PolicyUpdater system is its lack of expressive power to represent time-dependent authorisations. Consider the following authorisation rule:

> Bob holds $read$ access to file f_1 between $9:00\ AM$ and $5:00\ PM$.

The authorisation information above can be broken down into two parts: an authorisation part, i.e. "Bob holds read access to file f_1", and a temporal part, i.e. "between 9:00 AM and 5:00 PM". As language $\mathcal{L}$ can already express authorisations, we focus our attention to the temporal part. A naive attempt to extend language $\mathcal{L}$ to express time may involve adding two extra parameters to each authorisation atom to represent the starting and ending time points of the interval. For example, the authorisation rule above can be represented as:

$holds(bob, read, f_1, 900, 1700)$

The atom above may be interpreted to mean that the authorisation holds for all times between 9:00 AM and 5:00 PM, inclusive. In this example, the granularity of time, or the smallest unit of time that can be expressed, is one minute. Of course, a more general approach is to use the domain of positive integers. With this approach, the system can handle different granularities of time, where the choice of what time unit each discrete value is interpreted as is left to the application. For example, if the temporal values are defined to be the number of seconds since 12 midnight, 01 Jan 1970 (i.e. the UNIX epoch), then the atom below states that the authorisation holds at an interval starting at 9:00 AM, 18 March 1976 and ending at 5:00 PM, 18 March 1976:

$holds(bob, read, f_1, 195951600, 195980400)$

While this approach gives the language enough expressive power to represent authorisations bound by literal time values, it is by no means expressive enough to model relationships between the temporal intervals themselves. This deficiency is shown in the example below:

> $Alice$ holds a $write$ access right to file f_1 after Bob holds a $read$ access right to file f_2.

Such authorisation rule might arise in a scenario where the access right $write$ to file f_1 can only be granted in some time after the $read$ access right to file f_2 has been granted and revoked. This example shows that the specific times at which authorisations hold are not as important as the relationship between the times themselves. This authorisation rule may be represented as follows:

$holds(alice, write, f_1, i_1)$
$holds(bob, read, f_2, i_2)$
$after(i_1, i_2)$

The example above states that $alice$ holds a $write$ access right to file f_1 at some time interval i_1, bob holds a $read$ access right to file f_2 at some time interval i_2, and that the interval i_1 occurs at some time after the interval i_1. As mentioned earlier, the actual values of the time interval variables i_1 and i_2 is not as important as the fact that the interval i_1 occurs after interval i_2.

Allen [1] found that a total of 13 possible disjoint relations may exist between any two temporal intervals: $before$, $after$, $during$, $contains$, $meets$, $met\ by$, $starts$, $started\ by$, $finishes$, $finished\ by$ and $equals$. Furthermore, as each of these temporal interval relations are disjoint, he proposed an algebra to represent a network of interval relations, which may be composed of partial or disjunctive interval relation information.

At the time of writing, the authors of this paper are working on a new authorisation language, $\mathcal{L}^{\mathcal{T}}$. This new language is an extension of language $\mathcal{L}$ with provisions to: (1) express authorisation rules that hold only on specified time intervals, and (2) allow the representation of temporal interval relations either under Allen's full interval algebra or one of its subalgebras [14].

7 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 authorisation language, language $\mathcal{L}$, for the definition, updating and querying of access control policies. As we have shown, language $\mathcal{L}$ is expressive enough to represent constraints and default rules.

The case study in Section 5 demonstrated how the PolicyUpdater system can be adapted to be used in a real-world web server authorisation application. As mentioned earlier, while other logic based access control approaches have been proposed recently, most of these cannot deal with dynamic policy updates. Furthermore, most of these approaches do not address issues concerning implementation. To the best of our knowledge, the PolicyUpdater system is the first fully-implemented logic based access control system to be used in a web server security application.

Finally, as discussed in Section 6, we are currently working on extending language $\mathcal{L}$ and the PolicyUpdater system to support time-bound authorisation policies.

References

1. Allen J. F. (1983) Maintaining Knowledge about Temporal Intervals. Communications of the ACM, 26(11):832-843
2. Apache Software Foundation (2004) Authentication, Authorization and Access Control. Apache HTTP Server Version 2.1 Documentation, `http://httpd.apache.org/docs-2.1/`
3. Bai Y., Varadharajan V. (1999) On Formal Languages for Sequences of Authorization Transformations. In: Proceedings of Safety, Reliability and Security of Computer Systems (Lecture Notes in Computer Science), 1698:375-384

4. Bai Y., Varadharajan V. (2003) On Transformation of Authorization Policies. Data and Knowledge Engineering, 45(3):333-357
5. Baral C. (2003) Knowledge, Representation, Reasoning and Declarative Problem Solving. pp. 99-100, Cambridge University Press, UK
6. Bertino E., Buccafurri F., Ferrari E. Rullo P. (2000) A Logic-based Approach for Enforcing Access Control. Journal of Computer Security, 8(2-3):109-140
7. Bertino E., Mileo A., Provetti A. (2003) Policy Monitoring with User-Preferences in PDL. In: Proceedings of IJCAI-03 Workshop for Nonmonotonic Reasoning, Action and Change, pp. 37-44
8. Chomicki J., Lobo J., Naqvi S. (2000) 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
9. Crescini V. F., Zhang Y. (2004) A Logic Based Approach for Dynamic Access Control. In: Proceedings of the 17th Australian Joint Conference on Artificial Intelligence, 3339:623-635
10. Crescini V. F., Zhang Y., Wang W. (2004) Web Server Authorisation with the PolicyUpdater Access Control System. In: Proceedings of the 2004 IADIS WWW/Internet Conference, 2:945-948
11. Gelfond M., Lifschitz V. (1998) The Stable Model Semantics for Logic Programming. In: Proceedings of the Fifth International Conference on Logic Programming, pp. 1070-1080
12. Halpern J. Y., Weissman V. (2003) Using First-Order Logic to Reason About Policies. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop, pp. 187-201
13. Jajodia S., Samarati P., Sapino M. L., Subrahmanian V. S. (2001) Flexible Support for Multiple Access Control Policies. ACM Transactions on Database Systems, 29(2):214-260
14. Krokhin A., Jeavons P., Jonsson P. (2003) Reasoning about Temporal Relations: The Tractable Subalgebras of Allen's Interval Algebra. Journal of the ACM, 50(5):591-640
15. Laurie B., Laurie P. (2003) Apache: The Definitive Guide (3rd Edition). O'Reilly & Associates Inc., CA
16. Li N., Grosof B. N., Feigenbaum J. (2003) Delegation Logic: A Logic-Based Approach to Distributed Authorization. ACM Transactions on Information and System Security (TISSEC), 6(1):128-171
17. Lin F., Zhao X. (2004) On Odd and Even Cycles in Normal Logic Programs. In: Proceedings of AAAI 19th National Conference on Artificial Intelligence and 16th Conference on Innovative Applications of Artificial Intelligence, pp. 80
18. Lobo J., Bhatia R., Naqvi S. (1999) 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
19. Network Working Group (1999) HTTP 1.1 (RFC 2616). The Internet Society, `ftp://ftp.isi.edu/in-notes/rfc2616.txt`
20. Network Working Group (1999) HTTP Authentication: Basic and Digest Access Authentication (RFC 2617). The Internet Society, `ftp://ftp.isi.edu/in-notes/rfc2617.txt`
21. Simons P. (1995) Efficient Implementation of the Stable Model Semantics for Normal Logic Programs. Research Reports Number A35, Helsinki University of Technology, `http://www.tcs.hut.fi/Publications/reports/A35.ps.Z`

Appendix A Translation to Language $\mathcal{L}^*$

The following shows the language $\mathcal{L}^*$ translation of the language $\mathcal{L}$ program listing shown in Example 1.

1. Initial Fact Rules
   $\hat{memb}(alice, grp2, S_0) \leftarrow$
   $\hat{holds}(grp1, read, file, S_0) \leftarrow$
   $\hat{subst}(grp2, grp1, S_0) \leftarrow$
2. Constraint Rules
   $\hat{holds}(grp1, write, file, S_0) \leftarrow$
   $\quad \hat{holds}(grp1, read, file, S_0),$
   $\quad not\ \neg\hat{holds}(grp3, write, file, S_0)$
   $\hat{holds}(grp1, write, file, S_1) \leftarrow$
   $\quad \hat{holds}(grp1, read, file, S_1),$
   $\quad not\ \neg\hat{holds}(grp3, write, file, S_1)$
3. Policy Update Rules
   $\neg\hat{holds}(grp1, read, file, S_1) \leftarrow$
4. Inheritance Rules
   $\hat{holds}(alice, read, file, S_0) \leftarrow$
   $\quad \hat{holds}(grp1, read, file, S_0),$
   $\quad \hat{memb}(alice, grp1, S_0),$
   $\quad not\ \neg\hat{holds}(alice, read, file, S_0)$
   $\neg\hat{holds}(alice, read, file, S_0) \leftarrow$
   $\quad \neg\hat{holds}(grp1, read, file, S_0),$
   $\quad \hat{memb}(alice, grp1, S_0)$
   $\quad \vdots$
   $\hat{holds}(alice, write, file, S_1) \leftarrow$
   $\quad \hat{holds}(grp3, write, file, S_1),$
   $\quad \hat{memb}(alice, grp3, S_1),$
   $\quad not\ \neg\hat{holds}(alice, write, file, S_1)$
   $\neg\hat{holds}(alice, write, file, S_1) \leftarrow$
   $\quad \neg\hat{holds}(grp3, write, file, S_1),$
   $\quad \hat{memb}(alice, grp3, S_1)$
   $\hat{holds}(grp1, read, file, S_0) \leftarrow$
   $\quad \hat{holds}(grp2, read, file, S_0),$
   $\quad \hat{subst}(grp1, grp2, S_0)$
   $\neg\hat{holds}(grp1, read, file, S_0) \leftarrow$
   $\quad \neg\hat{holds}(grp2, read, file, S_0),$
   $\quad \hat{subst}(grp1, grp2, S_0)$
   $\quad \vdots$
   $\hat{holds}(grp3, write, file, S_1) \leftarrow$
   $\quad \hat{holds}(grp2, write, file, S_1),$
   $\quad \hat{subst}(grp3, grp2, S_1)$
   $\neg\hat{holds}(grp3, write, file, S_1) \leftarrow$
   $\quad \neg\hat{holds}(grp2, write, file, S_1),$
   $\quad \hat{subst}(grp3, grp2, S_1)$
5. Transitivity Rules
   $\hat{subst}(grp1, grp3, S_0) \leftarrow$
   $\quad \hat{subst}(grp1, grp2, S_0),\ \hat{subst}(grp2, grp3, S_0)$
   $\quad \vdots$
   $\hat{subst}(grp3, grp1, S_0) \leftarrow$
   $\quad \hat{subst}(grp3, grp2, S_0),\ \hat{subst}(grp2, grp1, S_0)$
   $\hat{subst}(grp1, grp3, S_1) \leftarrow$
   $\quad \hat{subst}(grp1, grp2, S_1),\ \hat{subst}(grp2, grp3, S_1)$
   $\quad \vdots$
   $\hat{subst}(grp3, grp1, S_1) \leftarrow$
   $\quad \hat{subst}(grp3, grp2, S_1),\ \hat{subst}(grp2, grp1, S_1)$
6. Inertial Rules
   $\hat{holds}(alice, read, file, S_1) \leftarrow$
   $\quad \hat{holds}(alice, read, file, S_0),$
   $\quad not\ \neg\hat{holds}(alice, read, file, S_1)$

$\neg \hat{holds}(alice, read, file, S_1) \leftarrow$
$\neg \hat{holds}(alice, read, file, S_0),$
$not\ \neg \hat{holds}(alice, read, file, S_1)$
$\hat{holds}(alice, write, file, S_1) \leftarrow$
$\hat{holds}(alice, write, file, S_0),$
$not\ \neg \hat{holds}(alice, write, file, S_1)$
$\neg \hat{holds}(alice, write, file, S_1) \leftarrow$
$\neg \hat{holds}(alice, write, file, S_0),$
$not\ \neg \hat{holds}(alice, write, file, S_1)$
$\hat{holds}(grp1, read, file, S_1) \leftarrow$
$\hat{holds}(grp1, read, file, S_0),$
$not\ \neg \hat{holds}(grp1, read, file, S_1)$
$\neg \hat{holds}(grp1, read, file, S_1) \leftarrow$
$\neg \hat{holds}(grp1, read, file, S_0),$
$not\ \neg \hat{holds}(grp1, read, file, S_1)$
$\vdots$
$\hat{holds}(grp3, read, file, S_1) \leftarrow$
$\hat{holds}(grp3, read, file, S_0),$
$not\ \neg \hat{holds}(grp3, read, file, S_1)$
$\neg \hat{holds}(grp3, read, file, S_1) \leftarrow$
$\neg \hat{holds}(grp3, read, file, S_0),$
$not\ \neg \hat{holds}(grp3, read, file, S_1)$
$\hat{holds}(grp1, write, file, S_1) \leftarrow$
$\hat{holds}(grp1, write, file, S_0),$
$not\ \neg \hat{holds}(grp1, write, file, S_1)$
$\neg \hat{holds}(grp1, write, file, S_1) \leftarrow$
$\neg \hat{holds}(grp1, write, file, S_0),$
$not\ \neg \hat{holds}(grp1, write, file, S_1)$
$\vdots$
$\hat{holds}(grp3, write, file, S_1) \leftarrow$
$\hat{holds}(grp3, write, file, S_0),$
$not\ \neg \hat{holds}(grp3, write, file, S_1)$
$\neg \hat{holds}(grp3, write, file, S_1) \leftarrow$
$\neg \hat{holds}(grp3, write, file, S_0),$
$not\ \neg \hat{holds}(grp3, write, file, S_1)$
$\hat{memb}(alice, grp1, S_1) \leftarrow$
$\hat{memb}(alice, grp1, S_0),$
$not\ \neg \hat{memb}(alice, grp1, S_1)$
$\neg \hat{memb}(alice, grp1, S_1) \leftarrow$
$\neg \hat{memb}(alice, grp1, S_0),$
$not\ \hat{memb}(alice, grp1, S_1)$
$\vdots$
$\hat{memb}(alice, grp3, S_1) \leftarrow$
$\hat{memb}(alice, grp3, S_0),$
$not\ \neg \hat{memb}(alice, grp3, S_1)$
$\neg \hat{memb}(alice, grp3, S_1) \leftarrow$
$\neg \hat{memb}(alice, grp3, S_0),$
$not\ \hat{memb}(alice, grp3, S_1)$
$\hat{subst}(grp1, grp1, S_1) \leftarrow$
$\hat{subst}(grp1, grp1, S_0),$
$not\ \neg \hat{subst}(grp1, grp1, S_1)$
$\neg \hat{subst}(grp1, grp1, S_1) \leftarrow$
$\neg \hat{memb}(grp1, grp1, S_0),$
$not\ \hat{memb}(grp1, grp1, S_1)$
$\vdots$
$\hat{subst}(grp3, grp3, S_1) \leftarrow$
$\hat{subst}(grp3, grp3, S_0),$
$not\ \neg \hat{subst}(grp3, grp3, S_1)$
$\neg \hat{subst}(grp3, grp3, S_1) \leftarrow$
$\neg \hat{memb}(grp3, grp3, S_0),$
$not\ \hat{memb}(grp3, grp3, S_1)$

7. Identity Rules

$\hat{subset}(grp1, grp1, S_0) \leftarrow$
$\hat{subset}(grp2, grp2, S_0) \leftarrow$
$\hat{subset}(grp3, grp3, S_0) \leftarrow$
$\hat{subset}(grp1, grp1, S_1) \leftarrow$
$\hat{subset}(grp2, grp2, S_1) \leftarrow$
$\hat{subset}(grp3, grp3, S_1) \leftarrow$

Appendix B Storage Structures

The data structures outlined in this section are used as a storage structure to hold the elements of language $\mathcal{L}$ before any operations are performed.

Each of the tables and lists used in the system inherits from a generic ordered and indexed list implementation. Each node in this list holds a generic data type that can be used to store strings, an arbitrary data type or another list type.

B.1 Symbol Table

The symbol table is used to store the identifier entities defined in the entity identifier declaration section of language $\mathcal{L}$ programs. The symbol table is composed of 6 separate string lists:

Field	Type	Description
ss	string list	single subject
sg	string list	group subject
as	string list	single access right
ag	string list	group access right
os	string list	single object
og	string list	group object

Each entity identifier are sorted in the above lists according to their type, and ordered according to the order in which they are declared in the program. Each list is indexed by consecutive positive integers starting from zero.

B.2 Policy Base

When a language $\mathcal{L}$ program is parsed, each of the facts, rules and policy updates must first be stored into the policy base. The policy base is composed of 4 tables to store the following: initial state facts, constraint rules, policy update definitions and the policy update sequence.

B.2.1 Atoms

The three types of atoms (holds, membership and subset) are represented as structures with 2 to 3 strings, with each string matching an entity identifier from the symbol table.

Atom	Field	Type	Description
holds	sub	string	subject entity
	acc	string	access right entity
	obj	string	object entity
member	elt	string	single entity
	grp	string	group entity
subset	$grp1$	string	subgroup entity
	$grp2$	string	supergroup entity

B.2.2 Facts

Facts are stored in a three-element structure composed of the following: polymorphic type which can be any of the three atom structures above; a type indicator to specify whether the fact is $holds$, $member$ or $subset$; and a truth flag, to indicate whether the atom is classically negated or not ($true$ if the fact holds and $false$ if the classical negation of the fact holds).

Field	Type	Description
$atom$	atom type	polymorphic structure
$type$	{h|m|s}	holds, member or subset
$truth$	boolean	negation indicator

B.2.3 Expressions

Since expressions are simply conjunctions of facts, they are represented as a list of fact structures.

B.2.4 Initial State Facts Table

The initial state facts table is represented as a single list of fact structures, or an expression. Each fact in all *initially* statements are added into the initial state facts table.

B.2.5 Constraint Table

The constraint table is represented as a list of constraint structures, with each structure composed of the following:

Field	Type	Description
exp	expression type	consequent
$pcond$	expression type	positive premise
$ncond$	expression type	negative premise

B.2.6 Policy Update Definition Table

Another list of structures is the policy update table. Each element structure of this table is composed of the following 4 fields:

Field	Type	Description
$name$	string	update identifier
$vlist$	ordered string list	variables
pre	expression type	precondition
$post$	expression type	postcondition

B.3 Policy Update Sequence Table

The policy update sequence table is an ordered list of sequence structures, each with the following elements:

Field	Type	Description
$name$	string	update identifier
$ilist$	ordered string list	identifiers