Implementation of a Logic-Based Access Control System with Dynamic Policy Updates and Temporal Constraints

Vino Fernando Crescini

A thesis submitted for the Degree of
Doctor of Philosophy at
University of Western Sydney

April 2006

Typeset in Times with TEX and LATEX 2_ε.

Except where otherwise indicated, this thesis is my own original work. I certify that this thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree.

V. F. Crescini
30 April 2006

For Huai Zheng

Acknowledgements

This work would not be possible without my principal supervisor, Assoc. Prof. Yan Zhang. For the countless hours I spent with him discussing the details of this work, for his guidance and for his support, I am deeply grateful. My gratitude also goes to Dr. Yun Bai, whose previous work on access control systems served as an initial motivation for this study.

This research was supported in part by an ARC Linkage-Projects Grant (LP0347878) in association with Smartlink Solutions Pty. Ltd. I thank Dr. Weiyuan Wang for his continued support for this project.

I would like to thank the academic staff and research colleagues in the Intelligent Systems Laboratory research group for their valuable inputs that contributed to this work.

I would also like to express my gratitude to my family, especially my parents, without whose encouragements and support this work would not be possible.

Finally, I thank my wife, Huai Zheng, the source of all my inspiration, the sole reason why I get up in the morning and do what I do.

Abstract

As information systems evolve to cope with the ever increasing demand of today's digital world, so does the need for more effective means of protecting information. In the early days of computing, information security started out as a branch of information technology. Over the years, several advances in information security have been made and, as a result, it is now considered a discipline in its own right. The most fundamental function of information security is to ensure that information flows to authorised entities, and at the same time, prevent unauthorised entities from accessing the protected information. In a typical information system, an access control system provides this function.

Several advances in the field information security have produced several access control models and implementations. However, as information technology evolves, the need for a better access control system increases. This dissertation proposes an effective, yet flexible access control system: the *PolicyUpdater* access control system.

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

In this thesis, we show the underlying mechanisms that make up the PolicyUpdater system, including the theoretical foundations of its formal language, the system structure, a full discussion of implementation issues and a performance analysis.

Lastly, the thesis also proposes a non-trivial extension of the PolicyUpdater system that is capable of supporting temporal constraints. This is made possible by the integration of the well-established *Temporal Interval Algebra* into the extended authorisation language, language $\mathcal{L}^{\mathcal{T}}$, which can also be translated into a normal logic program for evaluation. The formalisation of this extension, together with the full implementation details, are included in this dissertation.

Contents

List of Figures

List of Tables

Summary of Works

The following is a list of publications derived from this study:

- Crescini V. F., Zhang Y.
  PolicyUpdater: A System for Dynamic Access Control
  International Journal of Information Security
  Vol. 5, No. 3, pp. 145-165
  2006

- Crescini V. F., Zhang Y.
  A Logic Based Approach for Dynamic Access Control
  Proceedings of the 17th Australian Joint Conference on Artificial Intelligence (AI 2004, LNCS/LNAI)
  Vol. 3339, pp. 623-635
  2004

- Crescini V. F., Zhang Y., Wang W.
  Web Server Authorisation with the PolicyUpdater Access Control System
  Proceedings of the IADIS International Conference (WWW/Internet 2004)
  Vol. 2, pp. 945-948
  2004

The following is a list of software packages written for this dissertation:

- PolicyUpdater Access Control System Package (Vlad)

  `http://www.scm.uws.edu.au/~jcrescin/projects/policyupdater/index.html`

- Temporal Reasoner Engine Library (Tribe)

  `http://www.scm.uws.edu.au/~jcrescin/projects/tribe/index.html`

Chapter 1

Introduction

1.1 General Background

Generally, the term access control refers to a mechanism by which access to resources, digital or otherwise, are restricted. This restriction is enforced in such a way that only authorised entities are allowed access to the resources protected by the mechanism, and that any other entities are not. Access control restrictions are typically expressed as an access control *policy*, which defines the rules that determine whether or not an entity is granted access to protected resources.

In the broadest sense of the term, an access control system is a collection of mechanisms that enforces access control policies. An access control system may be divided into two sub-systems: the *authentication* sub-system and the *authorisation* sub-system. The goal of authentication is to obtain and verify the identity of every entity that requests access to protected resources. This may be achieved by simple mechanisms like username/password verification, digital certificates, physical/digital keys, or more advanced methods like biometric authentication which include fingerprint verification, voice print verification and iris/retinal scanning.

In contrast, an authorisation system's purpose is to decide whether or not an authenticated entity is to be allowed access to resources or not. It is therefore in the authorisation sub-system where the access control policy is kept and maintained. This is the reason why the term access control is often used interchangeably with the term authorisation.

Another system that may be considered an access control sub-system is the *enforcement* system. The sole task of this system is to ensure that an authenticated and authorised entity is allowed access and that all other entities are denied access. While in some literatures the enforcement system is considered to be part of the authorisation sub-system, others place it outside the domain of access control altogether. In this view, the task of an access control system is limited to identifying (authentication) requesting entities and deciding whether to

grant or deny (authorisation) access requests. Enforcement of these decisions is left to an external enforcement system.

Figure 1.1 shows how each sub-system fits into the overall access control system.

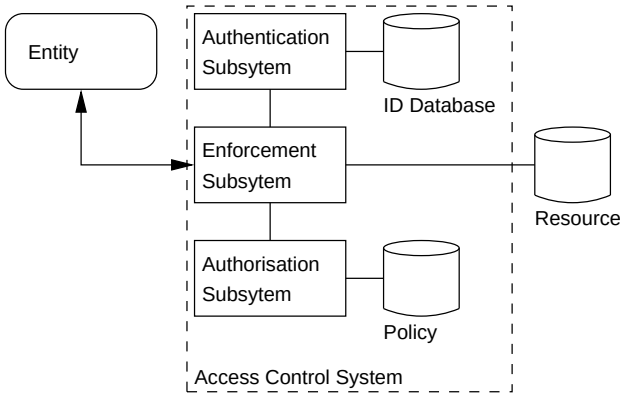

Figure 1.1: Structure of a Typical Access Control System

In this thesis, we shall focus on the authorisation sub-system, or more specifically, how the policies are expressed, read and maintained. It therefore assumes that the access control environment in which it is used provides adequate authentication and enforcement mechanisms.

1.1.1 Authorisation Rules

Example 1.1 shows a simple example of a partial policy that defines the authorisation rules of a file system. Note that in all three, a rule can be broken up into three general sorts: *subjects*, *access rights* and *objects*. In this example, the subjects are: $Alice$, Bob and $Charlie$; the access rights are: $read$, $write$ and $execute$; while the objects are: $file_0$, $file_1$ and $file_2$. From this example, it is easy to see what each sort is. Subjects are the entities which are granted or denied access to the resources, access rights are the types of privileges assigned to subjects for resources, and objects are the resources themselves. An *authorisation rule*, therefore, is a binding of subjects, access rights and object.

Example 1.1 *Below is an example of a partial file system policy:*

- $Alice$ *is allowed to* $read$ $file_0$
- Bob *is allowed to* $read$, $write$ *and* $execute$ $file_1$
- $Charlie$ *is allowed to* $read$ $file_0$, $file_1$ *and* $file_2$

1.1.2 Policy Base

A policy base is used by an authorisation system as a full repository of authorisation policies. Note that it is inaccurate to say that a policy base is simply a collection of authorisation rules. In Example 1.1 above, the three rules establishes the privileges granted to subjects $Alice$, Bob and $Charlie$. However, the set of three rules are incomplete: it provides no information as to what authorisations are allowed for $Dennis$, nor does it state that $Alice$ is not allowed to $write$ to $file_1$. A policy base must contain a full or complete policy.

To rectify the problem mentioned above, one solution might be to have the policy base adopt a closed-world assumption where any authorisation requests that are not explicitly addressed by a rule are rejected. The partial policy shown in Example 1.1 needs only a slight modification to make it complete:

Example 1.2 *A complete file system policy:*

- *Nobody is allowed any access to any resource, except:*
- $Alice$ *is allowed to* $read$ $file_0$
- Bob *is allowed to* $read$*,* $write$ *and* $execute$ $file_1$
- $Charlie$ *is allowed to* $read$ $file_0$*,* $file_1$ *and* $file_2$

1.2 Key Issues

In this section, we identify the key issues and requirements of the design and implementation of an authorisation system.

1.2.1 Formal Specification of Policies

The first issue to consider in an authorisation system is the structure of the policies. In the previous examples, the policies are defined as abstract descriptions of authorisation rules. However, in a real access control system, or to continue with the example, a real file system with hundreds of subjects and several levels of directories will require several pages of policy descriptions if expressed in this abstract form. A real authorisation system requires a formalised specification to express policies.

At the very least, a formal specification of an authorisation policy requires (1) a formal definition of entities (subjects, access rights and objects); (2) a formal specification of rules to bind together these entities to represent authorisation rules.

The first requirement deals with the issue of mapping authorisation entities, conceptual or otherwise, into formal representations which will be used in the definition of authorisation policies. In the previous example, this mapping is straight forward: system users to subjects, file permissions to access rights and files to objects. However, in some scenarios, entities may be mapped to something more abstract like "accounting documents" as objects or "administrators" as subjects. Generally, this requirement defines the mapping rules of entities as well as the formal definitions of the entities themselves.

The second requirement ensures that there is a formal method of expressing the rules themselves. In Example 1.1, the rules are expressed as a simple binding of the entities. However, as mentioned before, this example is only a partial policy since it does not state the rules that deal with other subjects or other objects. Example 1.2 rectified this problem, but the format of the first rule does not match the subject-access right-object binding format of the other three rules. This second requirement aims to formalise the definition of each rule.

For example, to formalise the policy specification of our file system policy, we define the following:

- An entity is an alphanumeric string.
- $Everybody$ is a special subject entity composed of all defined subjects, $AllAccess$ is an access right entity composed of all defined access rights, and $AllFiles$ is an object entity composed of all defined objects.
- An authorisation rule is composed of a boolean value to indicate whether the rule is a positive or a negative authorisation, followed by a binding of a subject, access right and object.
- In cases where two rules are in conflict, the most recent rule (between two rules, the one appearing lower in the list is more recent) overrides the other rule.

As shown above, the policy specification eliminates all ambiguities by providing a formal specification for each rule. Note that the positive and negative authorisation rule specification also defines how each rule is to be interpreted by the enforcement or access control system. Another issue handled by the above specification is how to interpret two conflicting rules. Example 1.3 shows the same policy used in the previous examples expressed in the above formalisation.

Example 1.3 *A formalised file system policy:*

- $False, Everyone, AllAccess, AllFiles$

- $True$, $Alice$, $read$, $file_0$
- $True$, Bob, $read$, $file_1$
- $True$, Bob, $write$, $file_1$
- $True$, Bob, $execute$, $file_1$
- $True$, $Charlie$, $read$, $file_0$
- $True$, $Charlie$, $read$, $file_1$
- $True$, $Charlie$, $read$, $file_2$

1.2.2 Policy Updates

Up to this point, we have been dealing only with static policies. A typical policy base, however, requires at least a means of changing the rules of the policy. A naive authorisation system implementation might only have the ability to handle static policies, but this would require a system reset every time the policy is changed. A robust authorisation system therefore needs a built-in mechanism that would allow its policy base to be changed or updated at run time. We call such updates that are performed at run time *dynamic updates*.

Continuing from our previous examples, suppose we need to add another file, say $file_3$, into the policy. With the current policy description, this file is "caught" by the negative authorisation rule which makes this file unreadable, unwritable and unexecutable by everyone. Now suppose we wish to make it readable by all current subjects, but not by any subject that might be added later. To do this, we need to add three new rules to the policy: one for $Alice$, one for Bob and another for $Charlie$. Similarly, if we wish to revoke the $write$ permission held by Bob for $file_1$, we simply update the policy base by deleting the appropriate rule.

While the policy updating scenario described above is simple, more complex update requirements do exist. One such requirement might arise in situations where a policy needs to be updated only when certain conditions are met. For example, we might require the subject $Alice$ to be granted a $write$ access right to $file_2$ if she already holds a $read$ access right to the same file. Such updates are called *conditional updates*.

1.2.3 Temporal Constraints

An authorisation rule composed only of the binding of a subject, an access right and an object represents a single authorisation that answers the question of *who* is allowed *what* permission to *which* resource. There is nothing to express *when* this authorisation rule is to hold. So far, in the examples given in the previous sections, we have made the assumption

that these rules apply at all times. That is, from some time in the past, either from the time the rules were defined or the time the access control system is activated, to the time the system is shut down. However, there are special situations where authorisation rules need to specify the time at which it is in effect as well as the usual subject-access right-object binding. Such situations might arise, for example, in a roster-based organisation where one user is granted access to some resource from 9AM to 2PM, another user from 2PM to 8PM and so forth.

This need to express time in authorisation rules can be easily satisfied by extending the rule specification to a quadruple binding of subjects, access rights, objects and time. For example, the rule ($Alice$, $read$, $file_0$, $1PM$ to $2PM$) might be used to represent a rule granting $Alice$ $read$ access to $file_0$ at lunch time. However, this approach is insufficient to express time-bound rules where the relationship between the time attributes is more important than the time attributes themselves. For example, this approach is unable to express the following abstract rule:

> $Alice$ is authorised to $read$ file $file_0$ only while Bob holds $read$ and $write$ access to the same file.

An authorisation system with support for temporal constraints must have its formal policy specification defined to express rules such as this.

1.2.4 Implementation

Obviously, a policy description is only as good as the authorisation system implementation on which it is used. The implementation of an authorisation system needs to address most importantly, the internal details of its policy base. The definition of the data structures to store entities and rules need to be considered, as well as the algorithms that make up the internal processes. Scalability is particularly important since a typical authorisation system policy is composed of a huge number of rules, and because each authorisation request usually requires instant response, efficiency of algorithms must also be taken into account.

As for the specification of policies, sufficient formalisation details must include, if, for example, formalised as a language, the syntax and semantics for implementation.

1.3 Literature Review

In this section, we review the different approaches to access control that have been proposed or implemented over the years.

1.3.1 Discretionary Access Control

The *Discretionary Access Control* (*DAC*) model is an authorisation model where policies are defined in such a way that a subject's identity determines what access rights it holds over which objects [16]. The two distinct characteristics of this model are: (1) every resource (object) in the system is owned by a subject; and (2) authorisation rules in the policy are bindings of subjects, access rights and objects. This access control model is discretionary in the sense that object owners (subjects) are capable of granting or revoking other subjects access rights to objects that they own at their discretion.

A very simple authorisation system based on the DAC model is the *access control matrix*, which was first proposed by Lampson [37], then subsequently extended in [29, 18, 31]. The access control matrix is a simple yet powerful policy base model where every subject's access rights of every object are stored. The matrix is composed of rows that represent the subjects of the system. Each column represents the objects of the system. The subject-object intersection, a cell, contains the access rights held by that subject to that object. An empty cell therefore means that the subject in that row do not possess any access rights to the object in that column. Table 1.1 shows an access control matrix with the same policy used in Example 1.3.

	$file_0$	$file_1$	$file_2$
$Alice$	r		
Bob		r,w,x	
$Charlie$	r	r	r
$Dennis$			

Table 1.1: An Example of an Access Control Matrix

As one might imagine, an access control matrix for a system composed mainly of user's personal data will be very sparse: only cells of intersecting objects and their owners will have access right entries stored in them. As a result, some applications of access control matrices take up more space that what is actually needed. Another implementation, the *access control list* [37], resolves this issue by storing authorisation rules in individual lists instead of one common matrix. In an access control list implementation, the access control system maintains a list of subjects with access rights for each object in the system. Access control lists are typically used to implement low level authorisation mechanisms like those used in file systems. It should be noted that very few systems actually implement an access control matrix. Indeed, the access control matrix was meant only as a conceptual representation of the policy base, and systems such as those based on access control lists are considered as implementations of the access control matrix model.

Figure 1.2 shows a set of access control lists that contains the same policy stored in the access control matrix in Table 1.1.

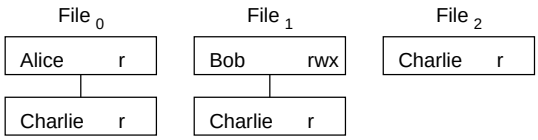

Figure 1.2: Access Control Lists

In some situations, it is more convenient to store authorisation policies in such a way that for each subject, the system maintains a *capability list* [23, 39] which contains all access rights the subject holds for certain objects. Since each subject has its own capability list, there might be cases where a certain subject's capability list is empty. This means the subject is not authorised to access any object. Figure 1.3 shows a set of capability lists that contains the same policy stored in the access control list shown in 1.2.

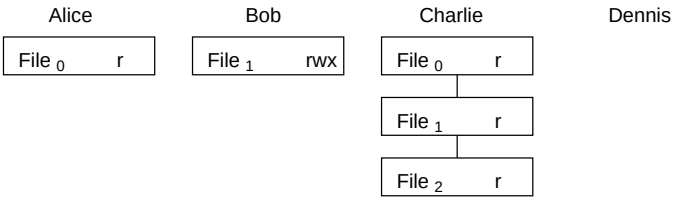

Figure 1.3: Capability Lists

1.3.2 Mandatory Access Control

The *Mandatory Access Control* (*MAC*) model introduces the concept of defining different levels of security. These levels are treated as security labels or attributes which represent the level of sensitivity when applied to objects, and define clearance level when applied to subjects. A classical example of security labels is the classification system formalised by Bell and LaPadula [9, 10] for the U.S. Department of Defense: (from least to most sensitive) unclassified, confidential, secret, top secret.

A subject is only allowed to read objects of equal or lower security level than the level assigned to the subject. Conversely, a subject can only write to objects that are of equal or higher security level than the level assigned to the subject. These two conditions are known as the *read down* and *write up* principles of MAC. They ensure that information can only be passed from a low security level to a high security level and not the other way.

It is mandatory in the sense that every subject and object must be assigned security labels, and as opposed to owner assignments of access rights in DAC, these assignments are made by administrators.

1.3.3 Role-Based Access Control

A recently proposed alternative to the DAC and MAC models is the *role-based access control* (*RBAC*) model. RBAC was first proposed in [24, 25], then later, a full RBAC framework in [55]. In most organisations, different roles are defined to achieve certain tasks. A person undertaking a particular role is granted access to all resources associated with that role. A person's authorisations gained from assuming a role are revoked once the person has relinquished the role or assumed another role. For example, in a hotel environment, the management role is assigned access to the hotel's accounting files, payroll database etc., while the front desk role's authorisation is limited to the guest and rooms database. A person assuming the role of a hotel manager is granted access to the resources assigned to that role, but the moment the person drops that role to assume the role of a front desk officer, access to these managerial resources are also dropped.

The goal of the RBAC model is to closely reflect this organisational structure of policies into an authorisation system. In this model, authorisations to access objects are assigned to roles, not to subjects directly. Subjects are then assigned different roles that they are allowed to partake. The strength of the RBAC model lies in its ability to accurately model abstract authorisation policies in real-world organisations. The abstraction of authorisation rules from subjects allows better policy management because the assignment of roles to subjects is easier to perform than direct assignment of authorisations to subjects.

1.3.4 Logic-Based Approach

An effective approach to access control is the *logic-based access control* 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.

One such model was proposed by Abadi et al. in [1]. Their model is based on a modal logic language designed for access control. However, this language focuses mainly on the delegation of authorisations and the concept of roles, rather than the expression of authorisation policies.

In 1992, Woo and Lam proposed an authorisation approach based on the concept that the semantics of authorisation should be separated from the low-level and system-dependent implementation mechanisms [64, 65]. In this approach, the policy is expressed as a set of rules written in terms of a first-order logic authorisation language. In this language, an

atom is a propositional constant of the form $a(s, o)$ representing a binding of a subject s, an access right a and an object o. A *positive authorisation atom* is written as $a^+(s, o)$, while a *negative authorisation atom* is written as $a^-(s, o)$. As a first-order logic language, it includes the usual notions of variables, negation, conjunction and disjunction. A *literal* is an atom or its negation, while a *formula* is either a literal, a conjunction of literals or a disjunction of literals. *Rules* are composed of formulas written in the form shown below:

$\frac{f:f'}{g}$

where

f is the prerequisite

f' is the assumption

g is the consequent

The rule above is interpreted as the following statement: "If f holds, and there is no reason not to believe that f' also holds, then g is also believed to hold". This rule construct is similar to the *default construct* used in *default logic* [50] where the absence of proof may be used as a condition for a rule.

The rules are evaluated into a policy base. In this approach, the policy base is defined as a 4-tuple: (P^+, P^-, N^+, N^-) where each component is a set of subject-access right-object tuples. P^+ contains a set of tuples that explicitly grant authorisations. For example, to evaluate whether subject s is given access right a to object o, one must check whether $(s, a, o) \in P^+$ is true. N^+ contains a set of tuples that explicitly deny authorisations. P^- and N^- records authorisations that should not be explicitly granted or denied, respectively.

Because the Woo and Lam approach explicitly expresses both positive and negative authorisations as well as conditional rules, it is not difficult to see that this approach provides a more flexible means of expressing authorisation policies. Example 1.4 shows the logic-based equivalent of the policy represented by the access control matrix shown in Table 1.1.

Example 1.4 *In this example, the first 7 rules explicitly grant different authorisations to specific subjects for specific objects. The final rule makes use of variables to deny all authorisations that are not explicitly granted.*

$\frac{True:True}{r^+(Alice,file_0)}$

$\frac{True:True}{r^+(Bob,file_1)}$

$\frac{True:True}{w^+(Bob,file_1)}$

$\frac{True:True}{x^+(Bob,file_1)}$

$$\frac{True{:}True}{r^{+}(Charlie, file_0)}$$

$$\frac{True{:}True}{r^{+}(Charlie, file_1)}$$

$$\frac{True{:}True}{r^{+}(Charlie, file_2)}$$

$$\frac{True{:}A^{-}(S,O)}{A^{-}(S,O)}$$

Note that the Woo and Lam model uses a single policy description for authorisation. Indeed, several authorisation models and their respective access control mechanism implementations [16] operate under a single authorisation policy. Such systems usually have this policy intertwined with its authorisation mechanism. As a consequence, such systems are limited to one specified authorisation policy, even though the authorisation requirement may change over time.

In response to this problem, Jajodia et al. proposed a logic-based authorisation language, the *Authorization Specification Language* [32]. Later, the approach was generalised into a full authorisation framework, the *Flexible Authorisation Framework* [33, 34]. With this framework, it is possible to enforce multiple authorisation policies within a single authorisation system. The framework itself is based on the logic-based authorisation language, through which administrators are allowed to choose, at their discretion, which policy is to be used. Other features of this framework include support for groups and roles, conflict resolution mechanisms and support for different decision strategies.

Another approach, proposed by Bertino et al. [13], uses an authorisation mechanism based on ordered logic. This powerful mechanism supports both positive and negative authorisations as well as implicit 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.

These systems, although effective, lack the details necessary to address the issues involved in the implementation of such systems. Another logic-based authorisation system was proposed by Halpern and Weissman [30]. Their approach focuses on the tractability of the policy reasoning mechanism by restricting the expressiveness of the policy language. This is achieved by using only a subset of first-order logic in the policy language. However, this tradeoff means that although the language provides sufficient expressive power for most applications, this approach cannot handle default propositions or policy updates.

1.3.5 Other Approaches and Considerations

The *Policy Description Language*, or *PDL*, developed by Lobo et al. [42], is a language for representing event and action oriented generic policies. *PDL* is later extended by Chomicki et al. [17] to include *policy monitors* which, in effect, are policy constraints. Bertino et

al. [14, 15], 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.

Sandhu et al. [54, 56, 57] was first to introduce the concept of *transformation*, or more specifically, *non-monotonic transformation* of access rights. For all intents and purposes, the term transformation is synonymous with our definition of policy updating. The non-monotonic property allows access rights to be revoked as well as granted in a policy update. Meadows [44], formalised the concept of dynamic upgrading of policies.

Bai and Varadharajan [5, 6, 7] extended the concept of dynamic updates by developing an authorisation model based on a language that is not only capable of handling policy updates, but also sequences of policy updates. Because this model is logic-based, other key features include the ability to express default propositions and a mechanism for conflict resolution. However, the major weakness of this framework is its lack of sufficient details for full system implementation.

Other access control approaches such as those proposed in [51, 52, 40] focus on authorisation delegation. Although such systems are useful in distributed or decentralised systems, a common deficiency is their inability to handle default propositions and/or policy updates.

Recently, Ray [49] proposed a formalism for updating access control policies in real-time. Although this formalism provides a set of algorithms aimed at the implementation of an authorisation system capable of performing dynamic policy updates, it only considers simple authorisation policies with no support for negative authorisations, conditional rules or conflict resolution.

Bertino et al. introduced an authorisation model in [11] that is capable of expressing temporal information in its policies. By including time interval attributes to rules, the model allows authorisations to be bound to a specific time period, or hold only for a specific time interval. In addition to this, the model also supports derivation rules based on temporal propositions. This feature allows new temporal authorisations to be valid on the basis of the presence or absence of other temporal authorisations. This model was later extended and revised in [12] to support periodic or cyclic temporal authorisations. In these models, the temporal derivation rules consider the validity of authorisations that are bound by time. However, the model cannot express relations between the time intervals themselves, as discussed in Section 1.2.3.

Ruan et al. [53] developed another logic-based authorisation model with support for delegation of temporal authorisations. Again, the main weakness of this model is that relations between time intervals cannot be expressed. Another authorisation model proposed by Atluri and Gal [4] focuses on securing web portals. In this model, the authorisations are

derived based on the temporal characteristics of the data (objects) and their relationships with other data. Although their work provides full implementation details, the model itself does not permit the assignment of temporal attributes to the authorisation rules.

The Extensible Access Control Markup Language (XACML) [48] is an XML-based access control language ratified by the Organization for the Advancement of Structured Information Standards (OASIS) in 2003 (version 1.0) and in 2005 (version 2.0). XACML not only defines a generalised language for representing authorisation rules and policies but also a format for expressing query requests and replies. Because it is based on XML, the language provides a flexibility that can be applied to model many different authorisation requirements. The extensible nature of the language allows it to be adapted to suit new requirements. Because XACML is a widely accepted standard, several implementations such as that of Sun Microsystems[1] already exist. However, one of the major drawbacks of XACML is its inability to express logical rules the way logic-based authorisation languages can. Furthermore, although extensions can be made to support time-based authorisation policies, the language has no native support for expressing relations between time intervals.

1.4 About the Thesis

The work presented in this thesis focuses on the key issues of authorisation systems outlined in Section 1.2, and at the same time addresses the deficiencies and limitations of existing access control models and approaches discussed in the previous section. In a nutshell, the work revolves around the formalisation and implementation of a logic-based authorisation system, *PolicyUpdater*, with support for constraint rules with default propositions, dynamic and conditional policy updates and temporal constraints.

The rest of the thesis is organised as follows:

The aim of Chapter 2 is to provide a means to formally express high-level authorisation policies with a logic-based formal specification language, language $\mathcal{L}$. The primary strength of this language lies in its ability to express logical rules and policy updates, as discussed in Sections 1.2.1 and 1.2.2. The chapter discusses the language's syntax and semantics, and by doing so, addresses the underlying mechanisms by which high level authorisation policies are expressed and evaluated.

Chapter 3 introduces the PolicyUpdater authorisation system which uses language $\mathcal{L}$ to express policies. The chapter gives an overview of the PolicyUpdater system as a whole, with its internal and external components. The chapter also includes a few algorithms that outline the system processes. An experiment that shows the relationship between input size

[1] Sun Microsystems XACML implementation from `http://sunxacml.sourceforge.net`

and execution time is also discussed in this chapter. Finally, the chapter is concluded by a case study which describes the application of PolicyUpdater as the primary authorisation system of a web server.

Chapter 4 is divided into two parts: the first part describes an interval algebra used to express relationships between temporal intervals, while the second part introduces language $\mathcal{L}^{\mathcal{T}}$, another authorisation language. Language $\mathcal{L}^{\mathcal{T}}$ is a non-trivial extension of language $\mathcal{L}$ that has enough expressive power to assign temporal attributes to authorisation rules and, by the integration of the interval algebra, allows the representation of temporal interval relations. The chapter includes a description of the extended language's syntax and semantics, as well as a discussion on query evaluation, policy updates and consistency checking.

Chapter 5 provides a full comprehensive implementation discussion of the extended PolicyUpdater system, with the specification of data structures and algorithms. The first half of the chapter outlines the full implementation details of a temporal interval relation reasoner. The second half of the chapter then describes the integration of this interval relation reasoner into an authorisation engine, thereby outlining the internal mechanisms of the extended PolicyUpdater system not previously discussed in Chapter 3.

Finally, Chapter 6 summarises the contributions of the work and outlines several future research directions.

Note that parts of this dissertation have already been published in journals and conference papers [20, 19, 22]. Furthermore, a paper on the extended PolicyUpdater system [21] will be submitted soon.

Chapter 2

Logic-Based Authorisation Language

Language $\mathcal{L}$[1]is a first-order logic language that represents a policy base for an authorisation system. Two key features of the language are: (1) the language provides a means to conditionally and dynamically update the policy base and (2) the semantics of the language allows a logic-based evaluation of an updated policy base to support authorisation queries.

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. The full BNF specification of the language is shown in Section A.1.

Components of Language $\mathcal{L}$

Language $\mathcal{L}$ statements are made up of one or more 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:

  1. *Entity Identifiers* are used to represent constant entities that make up a logical atom. They are divided into three types, with each type divided further into the *singular* and *group* entity sub-types:

     (a) *Subjects*: e.g. alice, lecturers, user-group.

[1]The full language $\mathcal{L}$ specification was first introduced in [19].

(b) *Access Rights*: e.g. read, write, own.

(c) *Objects*: e.g. file, database, directory.

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

```
[a-z][a-zA-Z0-9_]
```

2. *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 declarations 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_]
```

3. *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 or more alphanumeric and underscore characters. The first character of a variable identifier indicates its type ("S" for subject, "A" for access right and "O" for object). If the second character is an "S", then the variable is a place-holder for a singular entity while a "G" indicates that it is a place-holder for a group entity. The following regular expression shows the syntax of variable identifiers:

```
[SAO][SG][a-zA-Z0-9_]
```

- **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:

1. *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>)
```

2. *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>)
```

3. *Subset.* The subset atom states that the group identifiers grp_0 and grp_1 are of the same types and that group grp_0 is a subset of the group grp_1.

```
subst(<grp-0>, <grp-1>)
```

- **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 comma "," character:

```
<fact-0> [, <fact-1> [, ...]]
```

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

2.1.1 Declaration Statements

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

- **Entity Identifier Declarations**

  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 Declarations**

  The initial facts of the policy base, those that hold before any policy updates are performed, are declared by using the following syntax:

```
initially <ground-exp>;
```

- **Constraint Declarations**

  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 exp_0 holds if expression exp_1 is true and there is no evidence that exp_2 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 exp_0, exp_1 and exp_2 may be non-ground expressions, which means an identifier occurring within these expressions may be a variable.

```
always <exp-0>
  [implied by <exp-0>
  [with absence <exp-1>]];
```

- **Policy Update Declarations**

  Before a policy update can be applied, it must first be declared by using the following syntax:

```
<update-id>([<var-id>[, ...]])
  causes <exp-0>
  [if <exp-1>];
```

  $upd\text{-}id$ is the policy update identifier to be used in referencing this policy update. The optional parameter $var\text{-}id$ is a list which contains the variable identifiers occurring in expressions exp_0 and exp_1 and will be eventually replaced by entity identifiers when the update is referenced. The postcondition expression exp_1 is an expression that will hold in the state after this update is applied. The expression exp_1 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 the update directive described in the following section.

2.1.2 Directive Statements

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

- **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.

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

     ```
     seq add <update-id>([<entity-id>[, ...]]);
     ```

     where $update\text{-}id$ is the identifier of a declared policy update and the $entity\text{-}id$ list is a comma-separated list of entity identifiers that will replace the variable identifiers that occur in the declaration of the policy update.

  2. *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> <update-id>([<entity-id>[, ...]])
     ```

     where n is the ordinal index of the policy update in the sequence list starting at 0. $update\text{-}id$ is the policy update identifier and the $entity\text{-}id$ list is the comma-separated list of entity identifiers used to replace the variable identifier placeholders.

  3. *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>;
     ```

4. *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 Directives**

  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, to the initial state. Query expressions are answered with a *true*, *false* or *unknown*, depending on whether the queried expression holds, its negation holds, or neither, respectively. Syntax is as follows:

```
query <ground-exp>;
```

Example 2.1 *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 grp_2, which is a subset of group grp_1. The group grp_1 also initially holds a $read$ access right for the object $file$. The constraint states that if the group grp_1 has $read$ access for $file$, and no other information is present to indicate that grp_3 does not have $write$ access for $file$, then the group grp_1 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.

```
/* entity declarations */

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

/* initial fact statement */

initially
  memb(alice, grp2),
```

```
  holds(grp1, read, file),
  subst(grp2, grp1);

/* constraint statement */

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

/* policy update declaration */

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

/* add delete_read to policy update sequence list */

seq add delete_read(grp1, file);

compute;

/* queries */

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 [27]. The definition below formally defines the answer set of a logic program.

Definition 2.1 *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 2.2 *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.

$$PB \xrightarrow[u]{} PB'$$

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:

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

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

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

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.

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

Facts

A fact is a logical statement that makes a claim that an atom either holds or does not hold at a particular state. A fact that does not hold is said to be a *classically negated* fact [28]. 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 ρ0, ..., ρn;
```

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

$\hat{\rho}_0 \leftarrow$

$\vdots$

$\hat{\rho}_n \leftarrow$

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 $n_0, n_1, n_2 \geq 0$ may be represented as:

```
always ρ0_0, ..., ρ0_n0
  implied by ρ1_0, ..., ρ1_n1
  with absence ρ2_0, ..., ρ2_n2;
```

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 ρ_1 in the *implied by* clause as the positive premise and the expression ρ_2 in the *with absence* clause as the negative premise.

$\rho_{0_0} \leftarrow \rho_{1_0}, \ldots, \rho_{1_{n_1}}, not\ \rho_{2_0}, \ldots, not\ \rho_{2_{n_2}}$

$\vdots$

$\rho_{0_{n_0}} \leftarrow \rho_{1_0}, \ldots, \rho_{1_{n_1}}, not\ \rho_{2_0}, \ldots, not\ \rho_{2_{n_2}}$

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.

$\hat{\rho}_{0_0}^{S_0} \leftarrow \hat{\rho}_{1_0}^{S_0}, \ldots, \hat{\rho}_{1_{n_1}}^{S_0}, not\ \hat{\rho}_{2_0}^{S_0}, \ldots, not\ \hat{\rho}_{2_{n_2}}^{S_0}$

$\vdots$

$\hat{\rho}_{0_0}^{S_{|\psi|}} \leftarrow \hat{\rho}_{1_0}^{S_{|\psi|}}, \ldots, \hat{\rho}_{1_{n_1}}^{S_{|\psi|}}, not\ \hat{\rho}_{2_0}^{S_{|\psi|}}, \ldots, not\ \hat{\rho}_{2_{n_2}}^{S_{|\psi|}}$

$\vdots$

$\hat{\rho}^{S_0}_{0_{n_0}} \leftarrow \hat{\rho}^{S_0}_{1_0}, \ldots, \hat{\rho}^{S_0}_{1_{n_1}}, not\ \hat{\rho}^{S_0}_{2_0}, \ldots, not\ \hat{\rho}^{S_0}_{2_{n_2}}$

$\vdots$

$\hat{\rho}^{S_{|\psi|}}_{0_{n_0}} \leftarrow \hat{\rho}^{S_{|\psi|}}_{1_0}, \ldots, \hat{\rho}^{S_{|\psi|}}_{1_{n_1}}, not\ \hat{\rho}^{S_{|\psi|}}_{2_0}, \ldots, not\ \hat{\rho}^{S_{|\psi|}}_{2_{n_2}}$

where

$\hat{\rho}^{\sigma}_{0_i} = TransFact(\rho_{0_i}, \sigma), 0 \leq i \leq n_0,$

$\hat{\rho}^{\sigma}_{1_j} = TransFact(\rho_{1_j}, \sigma), 0 \leq j \leq n_1,$

$\hat{\rho}^{\sigma}_{2_k} = TransFact(\rho_{2_k}, \sigma), 0 \leq k \leq n_2,$

$S_0 \leq \sigma \leq S_{|\psi|}$

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

$n_0\ |\psi|$

where

n_0 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:

$\hat{holds}(alice, read, data, S_0) \leftarrow$

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

$\hat{holds}(alice, write, data, S_0) \leftarrow$

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

$\hat{holds}(alice, read, data, S_1) \leftarrow$

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

$\hat{holds}(alice, write, data, S_1) \leftarrow$

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

Policy Update Rules

Given that n_0, $n_1 \geq 0$, all occurrences of variable place-holders grounded to entity identifiers, a policy update u in language $\mathcal{L}$ is in the form:

u `causes` ρ_{0_0}`, ...,` $\rho_{0_{n_0}}$
`if` ρ_{1_0}`, ...,` $\rho_{1_{n_1}}$`;`

In language $\mathcal{L}^*$, such policy updates may be represented as a set of implications, with each fact ρ_0 in the postcondition expression as the consequent and precondition expression ρ_1 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.

$\hat{\rho}_{0_0} \leftarrow \hat{\rho}_{1_0}, \ldots, \hat{\rho}_{1_{n_1}}$

$\vdots$

$\hat{\rho}_{0_{n_0}} \leftarrow \hat{\rho}_{1_0}, \ldots, \hat{\rho}_{1_{n_1}}$

where

$\hat{\rho}_{0_i} = TransFact(\rho_{0_i}, Res(u, \sigma)), 0 \leq i \leq n_0,$

$\hat{\rho}_{1_j} = TransFact(\rho_{1_j}, \sigma), 0 \leq j \leq n_1$

Intuitively, a given language $\mathcal{L}$ policy update definition will generate n_0 policy update rules in language $\mathcal{L}^*$, where n_0 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 $\mathcal{L}^*$.

1. *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.

   (a) Subject Group Membership Inheritance Rules

   $\forall\, (ss, sg, a, o, \sigma),$

   $\hat{holds}(ss, a, o, \sigma) \leftarrow$
   $\quad \hat{holds}(sg, a, o, \sigma), \hat{memb}(ss, sg, \sigma), not\ \neg\hat{holds}(ss, a, o, \sigma)$

   $\neg\hat{holds}(ss, a, o, \sigma) \leftarrow$
   $\quad \neg\hat{holds}(sg, a, o, \sigma), \hat{memb}(ss, sg, \sigma)$

   where
   $\quad ss \in \mathcal{E}_{ss},$

$sg \in \mathcal{E}_{sg}$,
$a \in \mathcal{E}_a$,
$o \in \mathcal{E}_o$,
$S_0 \leq \sigma \leq S_{|\psi|}$

(b) Access Right Group Membership Inheritance Rules

$\forall\, (s, as, ag, o, \sigma)$,

$\hat{holds}(s, as, o, \sigma) \leftarrow$
$\hat{holds}(s, ag, o, \sigma), \hat{memb}(as, ag, \sigma), not\ \neg\hat{holds}(s, as, o, \sigma)$

$\neg\hat{holds}(s, as, o, \sigma) \leftarrow$
$\neg\hat{holds}(s, ag, o, \sigma), \hat{memb}(as, ag, \sigma)$

where
$s \in \mathcal{E}_s$,
$as \in \mathcal{E}_{as}$,
$ag \in \mathcal{E}_{ag}$,
$o \in \mathcal{E}_o$,
$S_0 \leq \sigma \leq S_{|\psi|}$

(c) Object Group Membership Inheritance Rules

$\forall\, (s, a, os, og, \sigma)$,

$\hat{holds}(s, a, os, \sigma) \leftarrow$
$\hat{holds}(s, a, og, \sigma), \hat{memb}(os, og, \sigma), not\ \neg\hat{holds}(s, a, os, \sigma)$

$\neg\hat{holds}(s, a, os, \sigma) \leftarrow$
$\neg\hat{holds}(s, a, og, \sigma), \hat{memb}(os, og, \sigma)$

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

(d) Subject Group Subset Inheritance Rules

$\forall\, (sg_0, sg_1, a, o, \sigma)$,

$\hat{holds}(sg_0, a, o, \sigma) \leftarrow$
$\hat{holds}(sg_1, a, o, \sigma), \hat{subst}(sg_0, sg_1, \sigma), not\ \neg\hat{holds}(sg_0, a, o, \sigma)$

$\neg\hat{holds}(sg_0, a, o, \sigma) \leftarrow$
$\neg\hat{holds}(sg_1, a, o, \sigma), \hat{subst}(sg_0, sg_1, \sigma)$

where

$sg_0, sg_1 \in \mathcal{E}_{sg}$,
$a \in \mathcal{E}_a$,
$o \in \mathcal{E}_o$,
$sg_0 \neq sg_1$,
$S_0 \leq \sigma \leq S_{|\psi|}$

(e) Access Right Group Subset Inheritance Rules

$\forall\, (s, ag_0, ag_1, o, \sigma)$,

$\hat{holds}(s, ag_0, o, \sigma) \leftarrow$
$\hat{holds}(s, ag_1, o, \sigma), \hat{subst}(ag_0, ag_1, \sigma), not\ \neg\, \hat{holds}(s, ag_0, o, \sigma)$

$\neg\hat{holds}(s, ag_0, o, \sigma) \leftarrow$
$\neg\hat{holds}(s, ag_1, o, \sigma), \hat{subst}(ag_0, ag_1, \sigma)$

where

$s \in \mathcal{E}_s$,
$ag_0, ag_1 \in \mathcal{E}_{ag}$,
$o \in \mathcal{E}_o$,
$ag_0 \neq ag_1$,
$S_0 \leq \sigma \leq S_{|\psi|}$

(f) Object Group Subset Inheritance Rules

$\forall\, (s, a, og_0, og_1, \sigma)$,

$\hat{holds}(s, a, og_0, \sigma) \leftarrow$
$\hat{holds}(s, a, og_1, \sigma), \hat{subst}(og_0, og_1, \sigma), not\ \neg\hat{holds}(s, a, og_0, \sigma)$

$\neg\hat{holds}(s, a, og_0, \sigma) \leftarrow$
$\neg\hat{holds}(s, a, og_1, \sigma), \hat{subst}(og_0, og_1, \sigma)$

where

$s \in \mathcal{E}_s$,
$a \in \mathcal{E}_a$,
$og_0, og_1 \in \mathcal{E}_{og}$,
$og_0 \neq og_1$,
$S_0 \leq \sigma \leq S_{|\psi|}$

2. *Transitivity Rules.* Given three distinct groups g_0, g_1 and g_2. If g_0 is a subset of g_1 and g_1 is a subset of g_2, then g_0 must also be a subset of g_2. The following rules ensure that the transitive property of subject, access right and object groups holds:

(a) Subject Group Transitivity Rules

$\forall\ (sg_0, sg_1, sg_2, \sigma)$,

$\hat{subst}(sg_0, sg_2, \sigma) \leftarrow \hat{subst}(sg_0, sg_1, \sigma), \hat{subst}(sg_1, sg_2, \sigma)$

where

$sg_0, sg_1, sg_2 \in \mathcal{E}_{sg}$,

$sg_0 \neq sg_1 \neq sg_2$,

$S_0 \leq \sigma \leq S_{|\psi|}$

(b) Access Right Group Transitivity Rules

$\forall\ (ag_0, ag_1, ag_2, \sigma)$,

$\hat{subst}(ag_0, ag_2, \sigma) \leftarrow \hat{subst}(ag_0, ag_1, \sigma), \hat{subst}(ag_1, ag_2, \sigma)$

where

$ag_0, ag_1, ag_2 \in \mathcal{E}_{ag}$,

$ag_0 \neq ag_1 \neq ag_2$,

$S_0 \leq \sigma \leq S_{|\psi|}$

(c) Object Group Transitivity Rules

$\forall\ (og_0, og_1, og_2, \sigma)$,

$\hat{subst}(og_0, og_2, \sigma) \leftarrow \hat{subst}(og_0, og_1, \sigma), \hat{subst}(og_1, og_2, \sigma)$

where

$og_0, og_1, og_2 \in \mathcal{E}_{og}$,

$og_0 \neq og_1 \neq og_2$,

$S_0 \leq \sigma \leq S_{|\psi|}$

3. *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 defined 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

$\hat{\alpha} \in \mathcal{A}^{\sigma}$,

$u \in \psi$,

$\hat{\alpha}' = CopyAtom(\hat{\alpha}, Res(u, \sigma))$

4. *Reflexivity 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
   $g \in (\mathcal{E}_{sg} \cup \mathcal{E}_{ag} \cup \mathcal{E}_{og})$,
   $S_0 \leq \sigma \leq S_{|\psi|}$

Example 2.2 *The following shows the language $\mathcal{L}^*$ translation of the language $\mathcal{L}$ program listing shown in Example 2.1.*

1. *Initial Fact Rules*

   $\hat{memb}(alice, grp_2, S_0) \leftarrow$
   $\hat{holds}(grp_1, read, file, S_0) \leftarrow$
   $\hat{subst}(grp_2, grp_1, S_0) \leftarrow$

2. *Constraint Rules*

   $\hat{holds}(grp_1, write, file, S_0) \leftarrow$
   $\quad \hat{holds}(grp_1, read, file, S_0)$, *not* $\neg\hat{holds}(grp_3, write, file, S_0)$

   $\hat{holds}(grp_1, write, file, S_1) \leftarrow$
   $\quad \hat{holds}(grp_1, read, file, S_1)$, *not* $\neg\hat{holds}(grp_3, write, file, S_1)$

3. *Policy Update Rules*

   $\neg\hat{holds}(grp_1, read, file, S_1) \leftarrow$

4. *Inheritance Rules*

   $\hat{holds}(alice, read, file, S_0) \leftarrow$
   $\quad \hat{holds}(grp_1, read, file, S_0)$, $\hat{memb}(alice, grp_1, S_0)$,
   $\quad$ *not* $\neg\hat{holds}(alice, read, file, S_0)$

   $\neg\hat{holds}(alice, read, file, S_0) \leftarrow$
   $\quad \neg\hat{holds}(grp_1, read, file, S_0)$, $\hat{memb}(alice, grp_1, S_0)$

$$\vdots$$

$$\hat{holds}(alice, write, file, S_1) \leftarrow$$
$$\hat{holds}(grp_3, write, file, S_1),\ \hat{memb}(alice, grp_3, S_1),$$
$$not\ \neg\hat{holds}(alice, write, file, S_1)$$

$$\neg\hat{holds}(alice, write, file, S_1) \leftarrow$$
$$\neg\hat{holds}(grp_3, write, file, S_1),\ \hat{memb}(alice, grp_3, S_1)$$

$$\hat{holds}(grp_1, read, file, S_0) \leftarrow$$
$$\hat{holds}(grp_2, read, file, S_0),\ \hat{subst}(grp_1, grp_2, S_0)$$
$$not\ \neg\hat{holds}(grp_1, read, file, S_0),$$

$$\neg\hat{holds}(grp_1, read, file, S_0) \leftarrow$$
$$\neg\hat{holds}(grp_2, read, file, S_0),\ \hat{subst}(grp_1, grp_2, S_0)$$

$$\vdots$$

$$\hat{holds}(grp_3, write, file, S_1) \leftarrow$$
$$\hat{holds}(grp_2, write, file, S_1),\ \hat{subst}(grp_3, grp2, S_1)$$
$$not\ \neg\hat{holds}(grp_3, write, file, S_1),$$

$$\neg\hat{holds}(grp_3, write, file, S_1) \leftarrow$$
$$\neg\hat{holds}(grp_2, write, file, S_1),\ \hat{subst}(grp_3, grp_2, S_1)$$

5. *Transitivity Rules*

$$\hat{subst}(grp_1, grp_3, S_0) \leftarrow \hat{subst}(grp_1, grp_2, S_0),\ \hat{subst}(grp_2, grp_3, S_0)$$

$$\vdots$$

$$\hat{subst}(grp_3, grp_1, S_0) \leftarrow \hat{subst}(grp_3, grp_2, S_0),\ \hat{subst}(grp_2, grp_1, S_0)$$

$$\hat{subst}(grp_1, grp_3, S_1) \leftarrow \hat{subst}(grp_1, grp_2, S_1),\ \hat{subst}(grp_2, grp_3, S_1)$$

$$\vdots$$

$$\hat{subst}(grp_3, grp_1, S_1) \leftarrow \hat{subst}(grp_3, grp_2, S_1),\ \hat{subst}(grp_2, grp_1, S_1)$$

6. *Inertial Rules*

$$\hat{holds}(alice, read, file, S_1) \leftarrow$$
$$\quad \hat{holds}(alice, read, file, S_0),\ not\ \neg\hat{holds}(alice, read, file, S_1)$$

$$\neg\hat{holds}(alice, read, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(alice, read, file, S_0),\ not\ \neg\hat{holds}(alice, read, file, S_1)$$

$$\hat{holds}(alice, write, file, S_1) \leftarrow$$
$$\quad \hat{holds}(alice, write, file, S_0),\ not\ \neg\hat{holds}(alice, write, file, S_1)$$

$$\neg\hat{holds}(alice, write, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(alice, write, file, S_0),\ not\ \neg\hat{holds}(alice, write, file, S_1)$$

$$\hat{holds}(grp_1, read, file, S_1) \leftarrow$$
$$\quad \hat{holds}(grp_1, read, file, S_0),\ not\ \neg\hat{holds}(grp1, read, file, S_1)$$

$$\neg\hat{holds}(grp_1, read, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(grp_1, read, file, S_0),\ not\ \neg\hat{holds}(grp_1, read, file, S_1)$$

$$\vdots$$

$$\hat{holds}(grp_3, read, file, S_1) \leftarrow$$
$$\quad \hat{holds}(grp_3, read, file, S_0),\ not\ \neg\hat{holds}(grp_3, read, file, S_1)$$

$$\neg\hat{holds}(grp_3, read, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(grp_3, read, file, S_0),\ not\ \neg\hat{holds}(grp_3, read, file, S_1)$$

$$\hat{holds}(grp_1, write, file, S_1) \leftarrow$$
$$\quad \hat{holds}(grp_1, write, file, S_0),\ not\ \neg\hat{holds}(grp_1, write, file, S_1)$$

$$\neg\hat{holds}(grp_1, write, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(grp_1, write, file, S_0),\ not\ \neg\hat{holds}(grp_1, write, file, S_1)$$

$$\vdots$$

$$\hat{holds}(grp_3, write, file, S_1) \leftarrow$$
$$\quad \hat{holds}(grp3, write, file, S_0),\ not\ \neg\hat{holds}(grp_3, write, file, S_1)$$

$$\neg\hat{holds}(grp_3, write, file, S_1) \leftarrow$$
$$\quad \neg\hat{holds}(grp_3, write, file, S_0),\ not\ \neg\hat{holds}(grp_3, write, file, S_1)$$

$$\hat{memb}(alice, grp_1, S_1) \leftarrow$$
$$\quad \hat{memb}(alice, grp_1, S_0),\ not\ \neg\hat{memb}(alice, grp_1, S_1)$$

$$\neg\hat{memb}(alice, grp_1, S_1) \leftarrow$$
$$\quad \neg\hat{memb}(alice, grp_1, S_0),\ not\ \hat{memb}(alice, grp_1, S_1)$$

$$\vdots$$

$$\hat{memb}(alice, grp_3, S_1) \leftarrow$$
$$\quad \hat{memb}(alice, grp_3, S_0),\ not\ \neg\hat{memb}(alice, grp_3, S_1)$$

$$\neg\hat{memb}(alice, grp_3, S_1) \leftarrow$$
$$\quad \neg\hat{memb}(alice, grp_3, S_0),\ not\ \hat{memb}(alice, grp_3, S_1)$$

$$\hat{subst}(grp1, grp_1, S_1) \leftarrow$$
$$\quad \hat{subst}(grp1, grp_1, S_0),\ not\ \neg\hat{subst}(grp_1, grp_1, S_1)$$

$$\neg\hat{subst}(grp_1, grp_1, S_1) \leftarrow$$
$$\quad \neg\hat{memb}(grp_1, grp_1, S_0),\ not\ \hat{memb}(grp_1, grp_1, S_1)$$

$$\vdots$$

$$\hat{subst}(grp_3, grp_3, S_1) \leftarrow$$
$$\quad \hat{subst}(grp_3, grp_3, S_0),\ not\ \neg\hat{subst}(grp_3, grp_3, S_1)$$

$$\neg\hat{subst}(grp_3, grp_3, S_1) \leftarrow$$
$$\quad \neg\hat{memb}(grp_3, grp_3, S_0),\ not\ \hat{memb}(grp_3, grp_3, S_1)$$

7. Reflexivity Rules

$$\hat{subset}(grp_1, grp_1, S_0) \leftarrow$$
$$\hat{subset}(grp_2, grp_2, S_0) \leftarrow$$
$$\hat{subset}(grp_3, grp_3, S_0) \leftarrow$$
$$\hat{subset}(grp_1, grp_1, S_1) \leftarrow$$
$$\hat{subset}(grp_2, grp_2, S_1) \leftarrow$$
$$\hat{subset}(grp_3, grp_3, S_1) \leftarrow$$

Definition 2.3 *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) reflexivity rules as described above.*

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 2.1 (Translation Size) *Given a domain description $\mathcal{D}_\mathcal{L}$; the sets Γ_{int}, Γ_{con} and Γ_{upd} 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 $Max(\Gamma_{int})$ in the expression of any* initially *statement in Γ_{int}; the maximum number of facts $Max(\Gamma_{con})$ in the* always *clause expression of any constraint statement in Γ_{con}; the maximum number of facts $Max(\Gamma_{upd})$ in the postcondition expression of any policy update statement in Γ_{upd}; and finally the policy update sequence list ψ, then the size of the translation $Trans(\mathcal{D}_\mathcal{L})$ is:*

$$
\begin{aligned}
&|Trans(\mathcal{D}_\mathcal{L})| \leq \\
&\quad Max(\Gamma_{int})\ |\Gamma_{int}| + \\
&\quad |\psi|\ Max(\Gamma_{con})\ |\Gamma_{con}| + \\
&\quad |\psi|\ Max(\Gamma_{upd}) + \\
&\quad 2\ |\psi|\ |\mathcal{E}_{ss}|\ |\mathcal{E}_{sg}|\ |\mathcal{E}_a|\ |\mathcal{E}_o| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_{as}|\ |\mathcal{E}_{ag}|\ |\mathcal{E}_o| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_a|\ |\mathcal{E}_{os}|\ |\mathcal{E}_{og}| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_{sg}|^2\ |\mathcal{E}_a|\ |\mathcal{E}_o| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_{ag}|^2\ |\mathcal{E}_o| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_s|\ |\mathcal{E}_a|\ |\mathcal{E}_{og}|^2 + \\
&\quad |\psi|\ (|\mathcal{E}_{sg}|^3 + |\mathcal{E}_{ag}|^3 + |\mathcal{E}_{og}|^3) + \\
&\quad 2\ |\psi|\ |\mathcal{A}| + \\
&\quad |\psi|\ (|\mathcal{E}_{sg}| + |\mathcal{E}_{ag}| + |\mathcal{E}_{og}|)
\end{aligned}
$$

Proof From Definition 2.3, it follows that the size of a language $\mathcal{L}^*$ translation is as follows:

$$|Trans(\mathcal{D}_{\mathcal{L}})| = |\mathcal{F}_{int}| + |\mathcal{F}_{con}| + |\mathcal{F}_{upd}| + |\mathcal{F}_{inh}| + |\mathcal{F}_{tra}| + |\mathcal{F}_{ine}| + |\mathcal{F}_{ref}|$$

where $\mathcal{F}_{int}$, $\mathcal{F}_{con}$, $\mathcal{F}_{upd}$, $\mathcal{F}_{inh}$, $\mathcal{F}_{tra}$, $\mathcal{F}_{ine}$, and $\mathcal{F}_{ref}$ are the sets of initial fact rules, constraint rules, policy update rules, inheritance rules, transitivity rules, inertial rules, and reflexivity rules, respectively.

As no *initially* statement in the set Γ_{int} contain an expression with more than $Max(\Gamma_{int})$ facts, the maximum number of initial fact rules generated in the translation is:

$$|\mathcal{F}_{int}| \leq Max(\Gamma_{int})\ |\Gamma_{int}|$$

Each language $\mathcal{L}$ constraint statement in Γ_{con} 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 $Max(\Gamma_{con})$ as the maximum number of facts in the *always* clause of any constraint statement, we have:

$$|\mathcal{F}_{con}| \leq |\psi|\ Max(\Gamma_{con})\ |\Gamma_{con}|$$

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

$$|\mathcal{F}_{upd}| \leq |\psi|\ Max(\Gamma_{upd})$$

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:

$$|\mathcal{F}_{inh}| = |\mathcal{F}_{inh_{mem}}| + |\mathcal{F}_{inh_{sub}}|$$

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:

$$\begin{aligned}
&|\mathcal{F}_{inh_{mem}}| = \\
&\quad 2\ |\psi|\ |\mathcal{E}_{ss}|\ |\mathcal{E}_{sg}|\ |\mathcal{E}_{a}|\ |\mathcal{E}_{o}| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_{s}|\ |\mathcal{E}_{as}|\ |\mathcal{E}_{ag}|\ |\mathcal{E}_{o}| + \\
&\quad 2\ |\psi|\ |\mathcal{E}_{s}|\ |\mathcal{E}_{a}|\ |\mathcal{E}_{os}|\ |\mathcal{E}_{og}|
\end{aligned}$$

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

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

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:

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

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:

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

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

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

■

2.3 Domain Consistency and Query 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[2], and how query evaluation is performed given a consistent language domain description. By using Definition 2.3, we define consistency as follows:

Definition 2.4 *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.*

[2]The strategy used here for determining domain consistency is based on a similar strategy used by Zhang [66] for characterising consistent action domains.

First, we introduce a few notational constructs. Given a domain description $\mathcal{D}_\mathcal{L}$ composed of the following language $\mathcal{L}$ statements:

```
initially ρ0_0, ..., ρ0_n0, !ρ1_0, ..., !ρ1_n1;

always ρ2_0, ..., ρ2_n2, !ρ3_0, ..., !ρ3_n3
  implied by ρ4_0, ..., ρ4_n4, !ρ5_0, ..., !ρ5_n5
  with absence ρ6_0, ..., ρ6_n6, !ρ7_0, ..., !ρ7_n7;

update u()
  causes ρ8_0, ..., ρ8_n8, !ρ9_0, ..., !ρ9_n9
  if ρ10_0, ..., ρ10_n10, !ρ11_0, ..., !ρ11_n11;
```

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:

$$\mathcal{F}^+_{int}(\gamma_{int}) = \{\rho_{0_i} \mid 0 \leq i \leq n_0\}$$
$$\mathcal{F}^-_{int}(\gamma_{int}) = \{\rho_{1_i} \mid 0 \leq i \leq n_1\}$$
$$\mathcal{F}^+_{con}(\gamma_{upd}) = \{\rho_{2_i} \mid 0 \leq i \leq n_2\}$$
$$\mathcal{F}^-_{con}(\gamma_{upd}) = \{\rho_{3_i} \mid 0 \leq i \leq n_3\}$$
$$\mathcal{F}^+_{upd}(\gamma_{con}) = \{\rho_{4_i} \mid 0 \leq i \leq n_4\}$$
$$\mathcal{F}^-_{upd}(\gamma_{con}) = \{\rho_{5_i} \mid 0 \leq i \leq n_5\}$$

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

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

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 *initially*, *constraint* or *policy update* declaration statement of language $\mathcal{L}$. We then define the following functions:

$$Eff(\gamma) = \begin{cases} \{\ \rho_{0_0}, \ldots, \rho_{0_{n_0}}, \neg\rho_{1_0}, \ldots, \neg\rho_{1_{n_1}}\ \}, \text{if } \gamma \text{ is } \textit{initially} \\ \{\ \rho_{2_0}, \ldots, \rho_{2_{n_2}}, \neg\rho_{3_0}, \ldots, \neg\rho_{3_{n_3}}\ \}, \text{if } \gamma \text{ is } \textit{constraint} \\ \{\ \rho_{8_0}, \ldots, \rho_{8_{n_8}}, \neg\rho_{9_0}, \ldots, \neg\rho_{9_{n_9}}\ \}, \text{if } \gamma \text{ is } \textit{policy update} \end{cases}$$

$$Def(\gamma) = \begin{cases} \emptyset, \text{if } \gamma \text{ is } \textit{initially} \\ \{\ \rho_{6_0}, \ldots, \rho_{6_{n_6}}, \neg\rho_{7_0}, \ldots, \neg\rho_{7_{n_7}}\ \}, \text{if } \gamma \text{ is } \textit{constraint} \\ \emptyset, \text{if } \gamma \text{ is } \textit{policy update} \end{cases}$$

$$Pre(\gamma) = \begin{cases} \emptyset, \text{if } \gamma \text{ is } \textit{initially} \\ \{\ \rho_{4_0}, \ldots, \rho_{4_{n_4}}, \neg\rho_{5_0}, \ldots, \neg\rho_{5_{n_5}}\ \}, \text{if } \gamma \text{ is } \textit{constraint} \\ \{\ \rho_{10_0}, \ldots, \rho_{10_0}, \neg\rho_{11_0}, \ldots, \neg\rho_{11_{n_1 1}}\ \}, \text{if } \gamma \text{ is } \textit{policy update} \end{cases}$$

Definition 2.5 *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

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

Stated simply, 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 2.6 *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 a policy update statement and $Eff(\gamma) = \overline{Eff(\gamma')}$.*

Definition 2.7 *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 2.2 (Domain Consistency) *A* **normal** *domain description of language $\mathcal{L}$ is also* **consistent**.

Proof From Definition 2.4, 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 2.7 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 [41]. 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 2.7, will generate an extended logic program $Trans(\mathcal{D}_\mathcal{L})$ without negative cycles. Also, from [8, 41], we further conclude that the translated program $Trans(\mathcal{D}_\mathcal{L})$ must have an answer set.

Condition 1 of Definition 2.7 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 2.7 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 2.7 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. ■

Since only consistent domain descriptions can be evaluated in terms of user queries, Theorem 2.2 may be used to check whether a domain description is consistent.

Definition 2.8 *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

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

Definition 2.8 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 2.3 *Given the language $\mathcal{L}$ code listing in Example 2.1 and its translation in Example 2.2, where the update sequence list ψ = $\{delete\_read(grp_1,\ file)\}$. The following shows the evaluated results of each query ϕ:*

ϕ_0 = $holds(grp_1, write, file) : TRUE$

ϕ_1 = $holds(grp_1, read, file) : FALSE$

ϕ_2 = $holds(alice, write, file) : TRUE$

ϕ_3 = $holds(alice, read, file) : FALSE$

2.4 Summary

In this chapter, we have provided a means of expressing access control policies in the form of Language $\mathcal{L}$, a first-order logic language. We have shown that the syntax of this language not only provides sufficient constructs to express logical rules in policies, but also a means of expressing conditional and dynamic update rules. More importantly, we have shown in this chapter that the semantics of Language $\mathcal{L}$ makes it possible to evaluate queries against a policy after an arbitrary sequence of policy updates.

The next chapter extends the ideas presented in this chapter by introducing a full access control system whose policies are expressed in Language $\mathcal{L}$.

Chapter 3

PolicyUpdater System

The PolicyUpdater system is a fully-implemented access control system that uses language $\mathcal{L}$ policy descriptions. The first part of this chapter focuses on the underlying mechanisms that make up the system, followed by an analysis of the performance of the implementation. The final part of the chapter describes the application of a PolicyUpdater module to a web server to enforce logic-based authorisation policies[1].

The source code and other technical information of the core PolicyUpdater system and the web server module can be found in the PolicyUpdater website at:

`http://www.scm.uws.edu.au/~jcrescin/projects/policyupdater/index.html`

3.1 System Structure

As shown in Figure 3.1, 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.

3.1.1 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.

[1] PolicyUpdater implementation and performance analysis were published in [20]. The web server module was first introduced in [22].

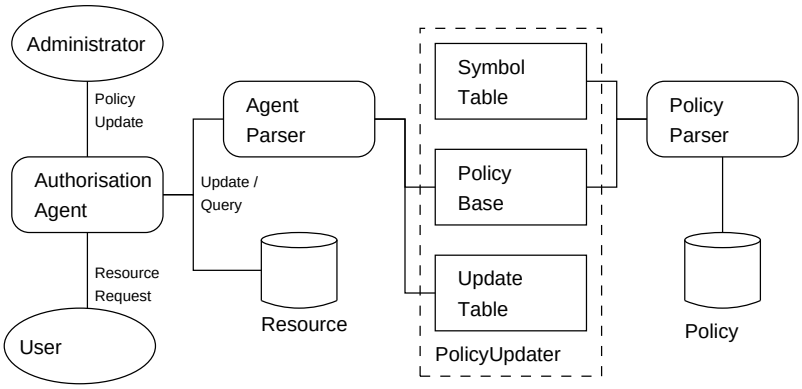

Figure 3.1: Structure of PolicyUpdater

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.

3.1.2 Data Structures

As a 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. This 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 in the policy base. The sequence of policy update directives are stored separately in the policy update sequence table.

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. This list structure implements the following operations:

- $LIST.Add(item)$ appends $item$ to the end of $LIST$.
- $LIST.Length()$ returns the number of elements in $LIST$.
- $LIST.Find(item)$ returns true if $item$ is in the list and false otherwise.
- $LIST.Index(item)$ returns the index of the item $item$ in $LIST$.
- $LIST.Get(index)$ returns the $index$'th item in $LIST$.

For simplicity, the following notations are used in the rest of this paper:

$|LIST|$ = $LIST.Length()$

$LIST[i]$ = $LIST.Get(i)$

Symbol Table

The symbol table is used to store the entity identifiers defined by the entity identifier declaration statements of language $\mathcal{L}$ programs. The symbol table is composed of 6 separate string lists as shown in 3.1.

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

Table 3.1: Symbol Table Data Structure

In addition to the 6 lists in the symbol table, 3 additional lists are defined: s, a and o. These lists are simple concatenations of the other lists in the table (i.e. s = $ss + sg$, etc.). Each entity identifier are sorted in the symbol table lists according to their type, and ordered according to the order in which they are declared in the program. Each list is indexed by positive integers starting from zero.

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 facts, constraint rules, policy update definitions and the policy update sequence.

Atoms. The three types of atoms, i.e. *holds*, *membership* and *subset*, are represented as structures of 2 to 3 strings, with each string matching an entity identifier from the symbol table. Table 3.2 shows the fields associated with each atom type.

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
	$type$	$\{sub\|acc\|obj\}$	Type Specifier
subset	grp_0	String	Subgroup Entity
	grp_1	String	Supergroup Entity
	$type$	$\{sub\|acc\|obj\}$	Type Specifier

Table 3.2: Atom Data Structure

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* type; and a truth flag to indicate whether the atom is classically negated or not (*true* if the fact holds and *false* if the negation of the fact holds). Table 3.3 shows the data structure for storing facts.

Field	**Type**	**Description**
$atom$	Atom Type	Polymorphic Structure
$type$	$\{h\|m\|s\}$	Holds, Member or Subset
$truth$	Boolean	Negation Indicator

Table 3.3: Fact Data Structure

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

Initial Facts Table. The initial 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 facts table.

Constraints Table. The constraints table, as shown in Table 3.4, is represented as a list of constraint structures, with each structure composed of three expression fields.

Field	Type	Description
exp	Expression Type	Consequent
pcn	Expression Type	Positive Premise
ncn	Expression Type	Negative Premise

Table 3.4: Constraints Table

Policy Update Definitions Table. Another list of structures is the policy update table. Each element structure of this table, as shown in Table 3.5 is composed of 4 fields.

Field	Type	Description
$name$	String	Update Identifier
$vlist$	Ordered String List	Variables
pre	Expression Type	Precondition
pst	Expression Type	Postcondition

Table 3.5: Policy Update Definitions Table

Policy Update Sequence Table

The policy update sequence table is an ordered list of sequence structures, each with two fields. Table 3.6 shows the structure of this table.

Field	Type	Description
$name$	String	Update Identifier
$ilist$	Ordered String List	Identifiers

Table 3.6: Policy Update Sequence Table

3.2 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 3.2 gives an overview of the system processes.

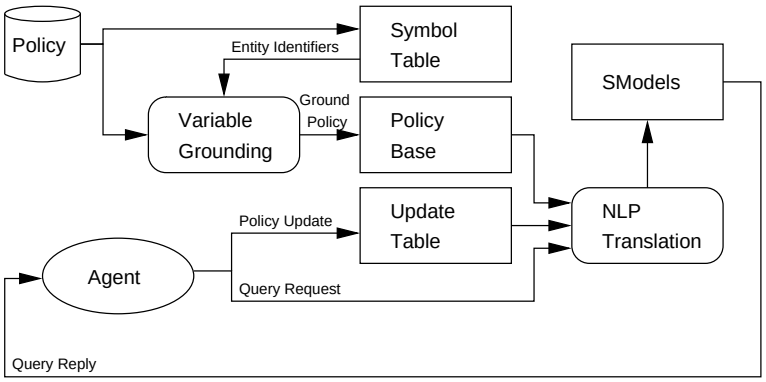

Figure 3.2: System Flowchart

3.2.1 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.

3.2.2 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.

3.2.3 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. This boolean parameter is used 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:

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

To ensure consistency, that is, to ensure that the fact and its negation are never both true at any one time, the following rule 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 described in Section 3.1.2, a normal logic program can then be generated for evaluation. The following algorithm generates a normal logic program, given the Symbol Table θ, Initial State Facts Table ω_i, Constraint Rules Table ω_c, Policy Update Definition Table ω_u, and Policy Update Sequence Table ψ:

Algorithm 3.1 *GenNLP()*

```
FUNCTION GenNLP(θ, ω_i, ω_c, ω_u, ψ)
  TransInitStateRules(ω_i)
  TransConstRules(ω_c, ψ)
  TransUpdateRules(ω_u, ψ)
  GenInherRules(θ, ψ)
  GenTransRules(θ, ψ)
  GenInertRules(θ, ψ)
  GenRefleRules(θ, ψ)
  GenConsiRules(θ, ψ)
ENDFUNCTION
```

The first three $Trans*$() functions in Algorithm 3.1 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.
- $RuleHead$(α) generates the consequent of the rule. α is a numeric representation of an atom (e.g. returned by the $Encode$() function).
- $RuleBody$(α, τ) generates the premise of the rule. α is an atom in numeric form like that of $RuleHead$(). τ is either $true$ or $false$, indicating whether the atom is positive or negative (negation-as-failure).
- $RuleEnd$() marks the end of a rule.

Algorithm 3.2 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.

Algorithm 3.2 *TransInitStateRules()*

```
FUNCTION TransInitStateRules(ω_i)
  FOR i = 0 TO (|ω_i| - 1) DO
    RuleBegin()
    RuleHead(Encode(ω_i[i].atom, 0, ω_i[i].truth))
    RuleBody(true, true)
    RuleEnd()
  ENDDO
ENDFUNCTION
```

The constraint rules generating algorithm (Algorithm 3.3) 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.

Algorithm 3.3 *TransConstRules()*

```
FUNCTION TransConstRules(ω_c, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|ω_c| - 1) DO
      FOR k = 0 TO (|ω_c[j].exp| - 1) DO
        RuleBegin()
        RuleHead(Encode(ω_c[j].exp[k].atom, i, ω_c[j].exp[k].truth))
        FOR k = 0 TO (|ω_c[j].pcn| - 1) DO
          α = Encode(ω_c[j].pcn[k].atom, i, ω_c[j].pcn[k].truth)
          RuleBody(α, true)
        ENDDO
        FOR k = 0 TO (|ω_c[j].ncn| - 1) DO
          α = Encode(ω_c[j].ncn[k].atom, i, ω_c[j].ncn[k].truth)
          RuleBody(α, false)
        ENDDO
        RuleEnd()
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

Algorithm 3.4 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.

Algorithm 3.4 *TransUpdateRules()*

```
FUNCTION TransUpdateRules(ω_u, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|ω_u| - 1) DO
      IF ψ[i].name == ω_u[j].name THEN
        upd = GndUpdate(ω_u[j], ψ[i].ilist)
        FOR k = 0 TO (|upd.pst| - 1) DO
          RuleBegin()
          RuleHead(Encode(upd.pst[k].atom, i + 1, upd.pst[k].truth))
          FOR l = 0 TO (|upd.pre| - 1) DO
            RuleBody(Encode(upd.pre[l].atom, i, upd.pre[l].truth), true)
          ENDDO
          RuleEnd()
        ENDDO
      ENDIF
    ENDDO
  ENDDO
ENDFUNCTION
```

The function $GndUpdate(u, ilist)$ used in Algorithm 3.4 returns a structure composed of two expressions pre and pst, which corresponds with the pre and pst 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 $ilist$.

Algorithm 3.5 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 constraint rule generating algorithm, each new rule generated is replicated for each policy update state.

Algorithm 3.5 *GenInherRules()*

```
FUNCTION GenInherRules(θ, ψ)
  GenSubSubstInherRules(θ, ψ)
```

```
  GenAccSubstInherRules(θ, ψ)
  GenObjSubstInherRules(θ, ψ)
  GenSubMembInherRules(θ, ψ)
  GenAccMembInherRules(θ, ψ)
  GenObjMembInherRules(θ, ψ)
ENDFUNCTION
```

The function $GenSubSubstInherRules()$ shown in Algorithm 3.6 generates the subject subset inheritance rules. Similar techniques are used for the generation of other subset and membership inheritance rules and are therefore not shown.

Algorithm 3.6 *GenSubSubstInherRules()*

```
FUNCTION GenSubSubstInherRules(θ, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.sg| - 1) DO
      FOR k = 0 TO (|θ.sg| - 1) DO
        IF θ.sg[j] != θ.sg[k] THEN
          FOR l = 0 TO (|θ.a| - 1) DO
            FOR m = 0 TO (|θ.o| - 1) DO
              α0 = holds(θ.sg[j], θ.ag[l], θ.og[m])
              α1 = holds(θ.sg[k], θ.ag[l], θ.og[m])
              α2 = subst(θ.sg[j], θ.sg[k])
              RuleBegin()
              RuleHead(Encode(α0, i, true))
              RuleBody(Encode(α1, i, true), true)
              RuleBody(Encode(α2, i, true), true)
              RuleBody(Encode(α0, i, false), false)
              RuleEnd()
              RuleBegin()
              RuleHead(Encode(α0, i, false))
              RuleBody(Encode(α1, i, false), true)
              RuleBody(Encode(α2, i, true), true)
              RuleEnd()
            ENDDO
          ENDDO
        ENDIF
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

Algorithm 3.7 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. As similar techniques are used to generate the access right and object transitivity rules, only the function that generates the subject transitivity rules is shown in Algorithm 3.8.

Algorithm 3.7 *GenTransRules()*

```
FUNCTION GenTransRules(θ, ψ)
  GenSubTransRules(θ, ψ)
  GenAccTransRules(θ, ψ)
  GenObjTransRules(θ, ψ)
ENDFUNCTION
```

Algorithm 3.8 *GenSubTransRules()*

```
FUNCTION GenSubTransRules(θ, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.sg| - 1) DO
      FOR k = 0 TO (|θ.sg| - 1) DO
        FOR l = 0 TO (|θ.sg| - 1) DO
          IF j != k AND j != l AND k != l THEN
            α0 = subst(θ.sg[j], θ.sg[l])
            α1 = subst({θ.sg[j], θ.sg[k])
            α2 = subst({θ.sg[k], θ.sg[l])
            RuleBegin()
            RuleHead(Encode(α0, i, true))
            RuleBody(Encode(α1, i, true), true)
            RuleBody(Encode(α2, i, true), true)
            RuleEnd()
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

The inertial rules generation function shown in Algorithm 3.9 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.

Algorithm 3.9 *GenInertRules()*

```
FUNCTION GenInertRules(θ, ψ)
  GenHldsInertRules(θ, ψ)
  GenMembInertRules(θ, ψ)
  GenSubsInertRules(θ, ψ)
ENDFUNCTION
```

Algorithm 3.10 shows the $GenHldsInertRules()$ function which generates the inertial rules for holds atoms. A similar method is used by the functions that generate the inertial rules for the other two atom types.

Algorithm 3.10 *GenHldsInertRules()*

```
FUNCTION GenHldsInertRules(θ, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.s| - 1) DO
      FOR k = 0 TO (|θ.a| - 1) DO
        FOR l = 0 TO (|θ.o| - 1) DO
          α = holds(θ.s[j], θ.a[k], θ.o[l])
          RuleBegin()
          RuleHead(Encode(α, i + 1, true))
          RuleBody(Encode(α, i, true), true)
          RuleBody(Encode(α, i + 1, false), false)
          RuleEnd()
          RuleBegin()
          RuleHead(Encode(α, i + 1, false))
          RuleBody(Encode(α, i, false), true)
          RuleBody(Encode(α, i + 1, true), false)
          RuleEnd()
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

The function $GenRefleRules()$ shown in Algorithm 3.11 generates the reflexivity 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. Algorithm 3.12 shows how the $GenSubRefleRules()$ function generates the reflexivity rules for subject groups.

Algorithm 3.11 *GenRefleRules()*

```
FUNCTION GenRefleRules(θ, ψ)
  GenSubRefleRules(θ, ψ)
  GenAccRefleRules(θ, ψ)
  GenObjRefleRules(θ, ψ)
ENDFUNCTION
```

Algorithm 3.12 *GenSubRefleRules()*

```
FUNCTION GenSubRefleRules(θ, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.sg| - 1) DO
      RuleBegin()
      RuleHead(Encode(subst(θ.sg[j], θ.sg[j]), i, true))
      RuleBody(true, true)
      RuleEnd()
    ENDDO
  ENDDO
ENDFUNCTION
```

The last two functions shown in Algorithm 3.13 and Algorithm 3.14 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.

Algorithm 3.13 *GenConsiRules()*

```
FUNCTION GenConsiRules(θ, ψ)
  GenHldsConsiRules(θ, ψ)
  GenMembConsiRules(θ, ψ)
  GenSubsConsiRules(θ, ψ)
ENDFUNCTION
```

Algorithm 3.14 *GenHldsConsiRules()*

```
FUNCTION GenHldsConsiRules(θ, ψ)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.s| - 1) DO
      FOR k = 0 TO (|θ.a| - 1) DO
        FOR l = 0 TO (|θ.o| - 1) DO
          α = holds(θ.s[j], θ.a[k], θ.o[l])
          RuleBegin()
          RuleHead(false)
          RuleBody(Encode(α, i, true), true)
          RuleBody(Encode(α, i, false), true)
          RuleEnd()
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDFUNCTION
```

3.2.4 Query Evaluation

Once a normal logic program has been generated from the authorisation policy stored in the storage structure, a set of answer sets may then be generated through the use of the stable model semantics [58, 47] with the *SModels* program[2]. 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.

Query evaluation works as follows. Given a set of answer sets Λ, an expression exp, the process of query evaluation returns either either *true*, *false* or *unknown* depending on the following conditions:

1. If each fact in the query expression exp is also in every answer set Λ, then exp is evaluated as *true*.
2. If the negation of each fact in exp is also in every answer set Λ, then exp is evaluated as *false*.
3. If none of the first two conditions are met, then exp is evaluated as *unknown*.

3.3 Experimental Results

In this section, we investigate the effects of domain size over computation time. The following tests[3]were conducted with PolicyUpdater version 1.0.5 using SModels version 2.31.

[2]SModels Package from `http://www.tcs.hut.fi/Software/smodels`

The test machine used is an AMD Athlon XP 1800+ PC with 1GB of RAM, running the Debian GNU/Linux 3.1 operating system with a plain Linux 2.6.16.20 kernel.

Table 3.7 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 3.7: Thirteen Test Cases with Different Domain Sizes

The language $\mathcal{L}$ code listing in Example 2.1 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 definitions 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.

Table 3.8 shows the execution times of each test case. T_C is the total time (in seconds) spent by the system to translate the language $\mathcal{L}$ 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 the accuracy of the results, each test was conducted 10 times. The

[3]The test results originally published in [20] was based on PolicyUpdater version 1.0.4 running on an older platform.

figures shown in Table 3.8 are the average times calculated from the 10 test runs.

	T_C	T_Q
1	0.000740	0.000616
2	0.249833	0.876428
3	0.071684	0.242012
4	13.344287	45.336900
5	0.251890	0.885376
6	0.253140	0.886952
7	0.250043	0.876544
8	14.302787	46.918752
9	0.249708	23.880376
10	14.374122	47.336968
11	14.499327	47.355992
12	14.509944	583.968320
13	-	-

Table 3.8: Average Computation Times in Seconds

As shown in Table 3.8, 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 2.2.3 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, particularly 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 5.615080 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 2.1.

3.4 Case Study: Web Server Application

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*[4]web server.

The Apache web server provides a generic access control system as provided by its *mod_auth* and *mod_access* modules [3, 38]. With this built-in access control system, Apache provides the standard HTTP *Basic* and *Digest* authentication schemes [46], 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 3.3, 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. 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

[4]Apache Web Server from `http://www.apache.org`

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.

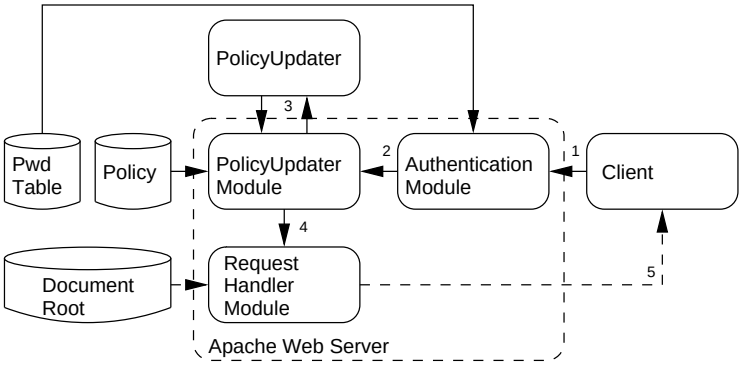

Figure 3.3: PolicyUpdater Module for the Apache Web Server

3.4.1 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 [45]: *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.

3.4.2 Mapping the Policy 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 declarations, constraint declarations and policy update declarations) are already in language $\mathcal{L}$ form.

3.4.3 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.

3.4.4 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.

Chapter 4

Temporal Constraints in Authorisation Policies

4.1 Introduction

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

Bob holds $read$ access to file f 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", 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:

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

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 beginning of 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:

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

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 time values themselves. This deficiency is shown in the example below:

$Alice$ holds a $write$ access right to file f_0 after Bob holds a $read$ access right to file f_1

Such authorisation rule might arise in a situation where the access right $write$ to file f_0 can only be granted at some time after the $read$ access right to file f_1 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:

$holds(alice, write, f_0, \iota_0)$

$holds(bob, read, f_0, \iota_1)$

$after(\iota_0, \iota_1)$

The example above states that $alice$ holds a $write$ access right to file f_0 at some time interval ι_0, bob holds a $read$ access right to file f_1 at some time interval ι_1, and that the interval ι_0 occurs at some time after the interval ι_1. As mentioned earlier, the actual values of the time intervals ι_0 and ι_1 is not as important as the fact that the interval ι_0 occurs after interval ι_1.

The rest of this chapter discusses how temporal constraints can be incorporated into the authorisation language. The next section introduces Allen's temporal interval algebra to express relations between time values, followed by a section that outlines extensions to this algebra, and finally, the last section gives a detailed formalisation of a new authorisation language that utilises the interval algebra to support temporal constraints.

4.2 Allen's Temporal Interval Algebra

Allen's interval algebra [2] is based on the fact that for any two well defined time intervals, there exists exactly one interval relation between them. The strength of the algebra lies not just on the formalisation of these relations, but also on its ability to handle disjunctive interval relations between undefined time intervals.

For example, given an interval ι_0, defined as 24th of September, 1995 to 25th of September, 1995; and interval ι_1, defined as 25th of December 1995 to 1st of January 1996. From

the definitions of these intervals, it is easy to conclude that interval ι_0 is *before* interval ι_1, since the finishing end point of interval ι_0 occurs before the starting end point of interval ι_1.

The strength of the interval algebra can be illustrated by considering intervals without endpoint definitions, but rather as having disjunctive relations with other intervals. For example, given three intervals ι_0, ι_1 and ι_2 whose bounding endpoints are not known. The algebra allows us to conclude that if ι_0 occurs either *before* or *during* ι_1 and ι_2 occurs *after* ι_1, then the interval ι_0 must occur *before* ι_2.

The rest of the section gives a detailed overview of Allen's interval algebra.

4.2.1 Time Points and Time Intervals

The preceding introduction hinted the difference between a time point and a time interval. This sub section aims to give a more formal distinction between the two.

A point in time represents an event with zero duration, or an event that occurs instantaneously, such as the switching on of a light bulb, or the moment the sun has risen in the morning.

A time interval, on the other hand, is defined to be the time that has elapsed between two given time points. For example, the interval one might call lunch hour may be defined as the time elapsed between the time points 1 PM and 2 PM.

Formally, a time interval ι is defined by its starting end point ι^- and a finishing end point ι^+, where $\iota^- < \iota^+$.

One might argue that time point events such as the switching on of a light bulb are not instantaneous, meaning that time, no matter how small the value, has elapsed between the instant that electricity flowed through light bulb's filament and the instant the light from the bulb reaches the observer's eyes. One can further argue that the "instant" that electricity flowed through the bulb's filament is not instantaneous, once we realise that the speed of light and electricity is finite. In other words, any given event can always be divided into sub-events. This argument is more evident in the sunrise example.

To solve this problem, Allen's algebra defines time intervals as the most basic entities. This means time points are not used to define time intervals, but instead, each interval is defined only in terms of its relationship with other intervals.

4.2.2 Time Interval Relations

As shown in Figure 4.1, the algebra defines 13 disjoint relations that can occur between any two intervals. For the sake of clarity, we define each temporal interval relation below in terms of the relationships of the end points of their intervals:

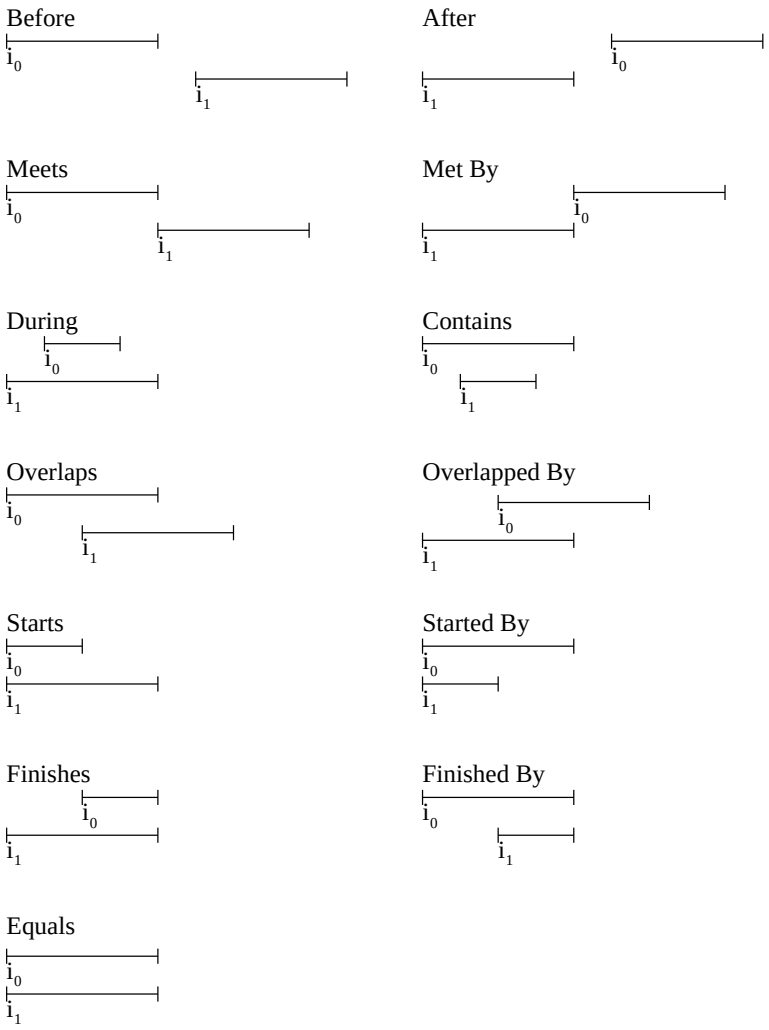

Figure 4.1: Thirteen Temporal Interval Relations

1. *Before*. Interval ι_0 is said to be *before* interval ι_1 if and only if $\iota_0^+ < \iota_1^-$.
2. *After*. Interval ι_0 is said to be *after* interval ι_1 if and only if $\iota_0^- > \iota_1^+$. Note that this is an inverse of the *before* relation.
3. *Meets*. Interval ι_0 is said to *meet* interval ι_1 if and only if $\iota_0^+ = \iota_1^-$.
4. *Met By*. Interval ι_0 is said to be *met by* interval ι_1 if and only if $\iota_0^- = \iota_1^+$. Note that this is an inverse of the *meets* relation.
5. *During*. Interval ι_0 is said to be *during* interval ι_1 if and only if $\iota_0^- > \iota_1^-$ and $\iota_0^+ < \iota_1^+$.
6. *Contains*. Interval ι_0 is said to *contain* interval ι_1 if and only if $\iota_0^- < \iota_1^-$ and $\iota_0^+ > \iota_1^+$. Note that this is an inverse of the *during* relation.
7. *Overlaps*. Interval ι_0 is said to *overlap* interval ι_1 if and only if $\iota_0^- < \iota_1^-$, $\iota_0^+ > \iota_1^-$ and $\iota_0^+ < \iota_1^+$.
8. *Overlapped By*. Interval ι_0 is said to be *overlapped by* interval ι_1 if and only if $\iota_0^- > \iota_1^-$, $\iota_0^- < \iota_1^+$ and $\iota_0^+ > \iota_1^+$. Note that this is an inverse of the *overlaps* relation.
9. *Starts*. Interval ι_0 is said to *start* interval ι_1 if and only if $\iota_0^- = \iota_1^-$ and $\iota_0^+ < \iota_1^+$.
10. *Started By*. Interval ι_0 is said to be *started by* interval ι_1 if and only if $\iota_0^- = \iota_1^-$ and $\iota_0^+ > \iota_1^+$. Note that this is an inverse of the the *starts* relation.
11. *Finishes*. Interval ι_0 is said to *finish* interval ι_1 if and only if $\iota_0^- > \iota_1^-$ and $\iota_0^+ = \iota_1^+$.
12. *Finished By*. Interval ι_0 is said to be *finished by* interval ι_1 if and only if $\iota_0^- < \iota_1^-$ and $\iota_0^+ = \iota_1^+$. Note that this is an inverse of the *finishes* relation.
13. *Equals*. Interval ι_0 *equals* interval ι_1 if and only if $\iota_0^- = \iota_1^-$ and $\iota_0^+ = \iota_1^+$.

For example, given intervals *lunch hour* and *work hours*, defined as 1 PM to 2 PM and 9 AM to 5 PM, respectively. Since 1 PM (1300 hours) is greater than 9 AM (0900 hours) and 2 PM (1400 hours) is less than 5 PM (1700 hours), then the interval *lunch hour* is *during* the interval *work hours*.

4.2.3 Inferring New Relations

As mentioned earlier, the strength of the algebra is its ability to infer new relations from existing ones. This is achieved by taking advantage of the transitive properties of relations. For example, given that interval ι_0 is *before* interval ι_1 and interval ι_1 is *before* interval ι_2. Regardless of what the end points are, interval ι_0 is *before* interval ι_2.

While simple relations like the one shown in the above example may seem intuitive, we quickly realise that it may not be so if we consider that a relation that exists between any two intervals may be given as a disjunctive set of possible relations. For example, the relation between interval ι_0 and interval ι_1 may be given as a set of possible relations $\{$*before*, *after*, *during*$\}$.

Another issue is propagation. Given that the temporal knowledge base is populated by these disjunctive interval relations, adding new pieces of information may narrow down the set of possible relations between two intervals. This in turn may lead to the trimming down of other relations between other interval pairs. In fact, as new and more specific information are added into the knowledge base, its effects may propagate to other relations.

Interval Relation Network

The temporal interval relation knowledge base is represented as a network whose nodes represent intervals and the arcs between them represent a set of possible relations that hold between the two intervals. Note that although this representation allows the assignment of a set of relations between any two intervals, because the relations are mutually exclusive, we know that only one of these relations actually holds. The fact that some interval pairs have a set of relations between them only suggests that the information given is insufficient to define the exact relation that holds between the intervals. Formally, we define the interval relation network as follows:

Definition 4.1 *A* **interval relation network** *is a collection of* **nodes** *and* **arcs***, where each node represents a single temporal interval and each arc represents a set of possible relations between two intervals. For any node pair, there is exactly one arc between them.*

The interval relation network is maintained in such a way that each node is connected to every other node in the network. In cases where no information is given to define an arc, we use the default arc which contains a set of all 13 relations. As a matter of convention in the notation, we only show one arc between two nodes. The reverse arc, composed of the inverses of the relations represented by the first arc, is omitted.

Figure 4.2 gives an example of a network with three nodes: ι_0, ι_1 and ι_2 and the following relations:

- Interval ι_0 is *before* or *during* interval ι_1.
- Interval ι_1 *overlaps* interval ι_2.

In this example, note that because no relation is defined for intervals ι_0 and ι_2, the corresponding arc between these nodes in Figure 4.2 is labelled *All*, meaning the relation set contains all 13 possible relations.

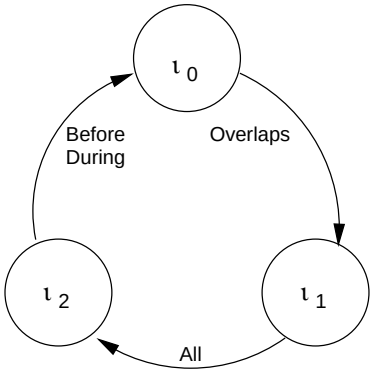

Figure 4.2: Network Representation Example

With the network structure defined, we can now formally define the three basic network operators:

Definition 4.2 *Given a temporal interval network NET. The three basic network operations are as follows:*

1. *The $NET.Get(\iota_0, \iota_1)$ operator returns the relation set on the arc between intervals ι_0 and ι_1.*
2. *The $NET.Replace(\iota_0, \iota_1, rs)$ operator replaces the relation set on the arc between intervals ι_0 and ι_1 with the relation set rs.*
3. *The $NET.AddRel(\iota_0, \iota_1, rs)$ operator adds the relation set rs to the arc between intervals ι_0 and ι_1, and propagates the effects of this change to the rest of the network.*

As described in the definition above, the $NET.Get()$ and $NET.Replace()$ operators are simple $get()$ and $set()$ operations on arc labels. In contrast, the $NET.AddRel()$ operator requires a more complex algorithm to perform the propagation. This algorithm is discussed in detail in the next section.

Propagation Algorithm

The algorithm works as follows. As a starting point, we assume that the network contains complete interval relation information, i.e. at each node, there is an arc that connects it to

	Before (BEF)	**After** (BEI)	**During** (DUR)	**Contains** (DUI)	**Overlaps** (OVR)	**Overlapped By** (OVI)	**Meets** (MET)	**Met** (MEI)	**Starts** (STA)	**Started By** (STI)	**Finishes** (FIN)	**Finished By** (FII)
Before (BEF)	BEF	ALL	BEF OVR MET DUR STA	BEF	BEF	BEF OVR MET DUR STA	BEF	BEF OVR MET DUR STA	BEF	BEF	BEF OVR MET DUR STA	BEF
After (BEI)	ALL	BEI	BEI OVI MEI DUR FIN	BEI	BEI OVI MEI DUR FIN	BEI	BEI OVI MEI DUR FIN	BEI	BEI OVI MEI DUR FIN	BEI	BEI	BEI
During (DUR)	BEF	BEI	DUR	ALL	BEF OVI MET DUR STI	BEI OVI MEI DUR FIN	BEF	BEI	DUR	BEI OVI MEI DUR FIN	DUR	BEF OVR MET DUR STA
Contains (DUI)	BEF MET DUI FII	BEI DUI MEI STI	OVR DUR STA FIN DUI STI FII EQL	DUI	OVR DUI FII	OVI DUI STI	OVR DUI FII	OVI DUI STI	DUI FII OVR	DUI	DUI STI OVI	DUI
Overlaps (OVR)	BEF	BEI OVI DUI MEI STI	OVR DUR STA	BEF OVR MET DUI FII	BEF OVR MET	OVR OVI DUR STA FIN DUI STI FII EQL	BEF	OVI DUI STI	OVR	DUI FII OVR	DUR STA OVR	BEF OVR MET
Overlapped By (OVI)	BEF OVR MET DUI FII	BEI	OVI DUR FIN	BEI OVI MEI DUI STI	OVR OVI DUR STA FIN DUI STI FII EQL	BEI OVI MEI	OVR DUI FII	BEI	OVI DUR FIN	OVI BEI MEI	OVI	OVI DUI STI
Meets (MET)	BEF	BEI OVI MEI DUI STI	OVR DUR STA	BEF	BEF	OVR DUR STA	BEF	FIN FII EQL	MET	MET	DUR STA OVR	BEF
Met By (MEI)	BEF OVR MET DUI FII	BEI	OVI DUR FIN	BEI	OVI DUR FIN	BEI	STA STI EQL	BEI	DUR FIN OVI	BEI	MEI	MEI
Starts (STA)	BEF	BEI	DUR	BEF OVR MET DUI FII	BEF OVR MET	OVI DUR FIN	BEF	MEI	STA	STA STI EQL	DUR	BEF MET OVR
Started By (STI)	BEF OVR MET DUI FII	BEI	OVI DUR FIN	DUI	OVR DUI FII	OVI	OVR DUI FII y	MEI	STA STI EQL	STI	OVI	DUI
Finishes (FIN)	BEF	BEI	DUR	BEI OVI MEI DUI STI	OVR DUR STA	BEI OVI MEI	MET	BEI	DUR	BEI OVI MEI	FIN	FIN FII EQL
Finished By (FII)	BEF	BEI OVI MEI DUI STI	OVR DUR STA	DUI	OVR	OVI DUI STI	MET	STI OVI DUI	OVR	DUI	FIN FII EQL	FII

Table 4.1: Transitivity Table

every other node. Where there is no defined relationship between two nodes, the default arc is used to connect these two nodes. The algorithm is invoked whenever new information is to be added into the network. Whenever a new relation is added into the network, all consequences of this new relation are also added into the network. These consequences are computed through the transitive closure of the network. The following example illustrates this procedure.

Given 3 intervals ι_0, ι_1 and ι_2, and the relation ι_0 is *before* ι_1. Suppose the new relation ι_2 is *during* ι_1 is added into the network. The algorithm then infers that ι_0 is *before* ι_2. This new relation is again added into the network in a similar way, possibly inferring other new relations as it is added. This procedure is repeated until no new information is yielded.

Table 4.1 shows the basic transitivity rules. For any 3 intervals ι_0, ι_1 and ι_2, the relation(s) between intervals ι_0 and ι_2 is shown in the intersection of the row that contains the relation between ι_0 and ι_1, and the column that contains the relation between ι_1 and ι_2.

Before we can define the actual algorithm, we must first define a few functions. For any single relations r_0 and r_1, the function $Trans_1(r_0, r_1)$ returns the relation set rs that corresponds to the intersection of r_0 and r_1 in Table 4.1.

Using this function, we can then define the extended function $Trans_2(rs_0, rs_1)$ shown in Algorithm 4.1, which takes a pair of relation sets rs_0 and rs_1 as input, and returns the relation set rs which contains all the possible relations inferred from the two given relation sets using the $Trans_1()$ function.

Algorithm 4.1 *Trans$_2$()*

```
FUNCTION Trans2(rs0, rs1)
  rs = ∅
  FOR each r0 ∈ rs0 DO
    FOR each r1 ∈ rs1 DO
      rs = rs ∪ Trans1(r0, r1)
    ENDDO
  ENDDO
  RETURN rs
ENDFUNCTION
```

In addition to these functions, we also define a standard queue structure Q which stores network arcs, i.e. a pair of intervals and a relation set that holds between the two intervals. Thus, we have three operators for the queue structure:

- $Q.Enqueue(\iota_0, \iota_1, RS)$ stores the given arc to the end of the structure.
- $Q.Dequeue()$ returns and removes the arc at the front of the structure.

- $Q.IsEmpty()$ returns true if the queue is empty and false otherwise.

The $NET.AddRel()$ algorithm shown in Algorithm 4.2 works as follows. Every time a new arc (ι_0, ι_1, rs) is added, the algorithm finds the transitive relation set RS between each of these intervals and every other interval I in the network, i.e. the algorithm finds $RS = Trans_2(NET.Get(\iota, \iota_0), rs)$ for every other interval I in the network. If this new relation set RS contains more specific information than what is already in the network, i.e. $RS \subset NET.Get(I, \iota_1)$, then this new arc between I and ι_1 (shown in Figure 4.3) is again put through the same algorithm, as it might yield more relation changes.

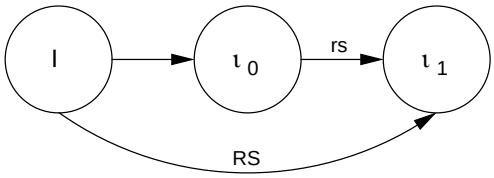

Figure 4.3: New Relation RS From Interval I and Interval ι_1

Note that the algorithm also attempts to form new transitive relations between the given intervals ι_0 and ι_1 and all other intervals I in the network such that interval I is to the right of the other two intervals (shown in Figure 4.4).

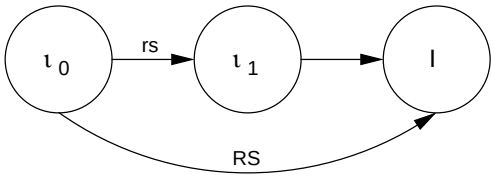

Figure 4.4: New Relation RS From Interval ι_0 and Interval I

Algorithm 4.2 *NET.AddRel()*

```
FUNCTION NET.AddRel(ι0, ι1, rs)
  Q.Enqueue(ι0, ι1, rs)
  WHILE NOT Q.IsEmpty() DO
    (ι'0, ι'1, rs') = Q.Dequeue()
    NET.Replace(ι'0, ι'1, rs')
    FOR each interval ι'' ∈ NET DO
      IF ι'' != ι'0 AND ι'' != ι'1 THEN
        rs'' = Trans2(NET.Get(ι'', ι'0), rs')
        rs''' = NET.Get(ι'', ι'1) ∩ rs''
```

```
        IF rs''' ⊂ NET.Get(ι'', ι'_1) THEN
          Q.Add(ι'', ι'_1, rs''')
        ENDIF
      ENDIF
    ENDDO
    FOR each interval ι'' ∈ NET DO
      IF ι'' != ι'_0 AND ι'' != ι'_1 THEN
        rs'' = Trans_2(rs', NET.Get(ι'_1, ι''))
        rs''' = NET.Get(ι'_0, ι'') ∩ rs''
        IF rs''' ⊂ NET.Get(ι'_0, ι'') THEN
          Q.Add(ι'_0, ι'', rs''')
        ENDIF
      ENDIF
    ENDDO
  ENDDO
ENDFUNCTION
```

Example 4.1 *Given a network with three intervals ι_0, ι_1 and ι_2 where no relation between any of the intervals are known. As no relations are given, each arc in the network as shown in Figure 4.5 is the default arc.*

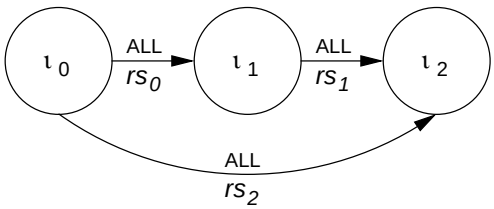

Figure 4.5: Network with 3 Default Arcs

Now, suppose the relation between interval ι_0 and interval ι_1 is narrowed down to the relation set $\{before, meets, overlaps\}$, i.e., the following operation is executed:

$NET.AddRel(\iota_0, \iota_1, \{BEF, MET, OVR\})$

This operation will yield the network shown in Figure 4.6. Note that because every arc in the network contains all relations, the effects of the $NET.AddRel()$ operation are limited to one arc.
Now, if the relation set $\{starts, finishes\}$ is added to the arc between interval ι_1 and interval ι_2:

$NET.AddRel(\iota_1, \iota_2, \{STA, FIN\})$

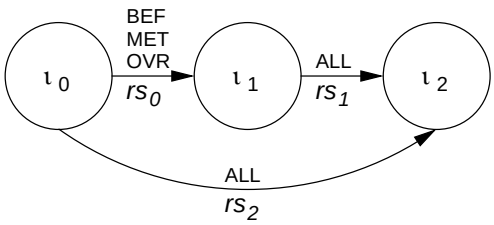

Figure 4.6: Network after $NET.AddRel(\iota_0, \iota_1, \{BEF, MET, OVR\})$

The algorithm will also compute the relation set rs_2 between interval ι_0 and interval ι_2 by using the relation set rs_0 between intervals ι_0 and interval ι_1, and relation set rs_1 between intervals ι_1 and interval ι_2 to complete the transitivity. In other words, the algorithm computes the following:

rs_2 = $Trans_2(rs_0, rs_1)$

rs_2 = $Trans_2(\{BEF, MET, OVR\}, \{STA, FIN\})$

By referring to the transitivity table in Table 4.1, we note the following:

$Trans_1(BEF, STA)$ = $\{BEF\}$

$Trans_1(BEF, FIN)$ = $\{BEF, OVR, MET, DUR, STA\}$

$Trans_1(MET, STA)$ = $\{MET\}$

$Trans_1(MET, FIN)$ = $\{DUR, STA, OVR\}$

$Trans_1(OVR, STA)$ = $\{OVR\}$

$Trans_1(OVR, FIN)$ = $\{DUR, STA, OVR\}$

Therefore, as $Trans_2()$ takes the union of all the relation sets returned by calls to $Trans_1()$, we have:

rs_2 = $\{BEF, OVR, MET, DUR, STA\}$

Finally, because rs_2 is a subset of $NET.Get(\iota_0, \iota_2)$, the algorithm replaces the arc between ι_0 and ι_2 with rs_2, as shown in Figure 4.7.

4.3 Extensions to Allen's Interval Algebra

In an authorisation system, an agent that enforces an authorisation policy must be able to match the policy's entities to objects observable by the agent, e.g. a subject in the policy

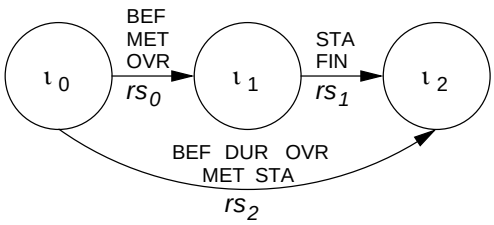

Figure 4.7: Network after $NET.AddRel(\iota_1, \iota_2, \{STA, FIN\})$

is mapped to a user that logs in, or an object in the policy is mapped to a certain file in the filesystem. Similarly, an agent must also be able to map time intervals in a policy to events that are observable by the agent. For example, the interval *logged_on* in the policy may be mapped to the time between a certain user logs in and out of the system.

In certain situations, however, it is convenient to map intervals in the policy to a particular time scale, such as the agent's real time clock. These situations might arise when certain authorisations need to be granted or revoked at a certain time, instead of being triggered by observable events. To map policy intervals to real time, it is necessary to define them in terms of points in real time.

In the previous section, we have shown that Allen's algebra defines temporal intervals as the primitive, where each interval is defined not by time points, but by their relationships with other intervals. In this section, we will attempt to show the extensions to the interval algebra to allow it to express intervals in terms of time points.

4.3.1 Time Points Revisited

By allowing time point definitions to be expressed in the algebra, we must formally define time points.

Disregarding the physical effects of gravitation and velocity on time, we make the assumption that time is linear, absolute and universal. That is, time always flows in one direction: from past to future; that the passage of time as seen by one system is the same for any other system; and that all systems define time against a universal frame of reference. For simplicity, we further assume that time is not a continuous line, but is instead made up of discrete time points.

As this definition goes against the more intuitive notion of continuous time lines, one might see the problem of events "falling through" the gaps between the discrete time points in the non-continuous time line.

To go around this problem, we allow the time point granularity to be chosen arbitrarily. Ideally, for a specific application, one would choose a granularity that is at least as small as

the smallest interval in that application. In practice, however, such choices are limited by implementing system's clock and other hardware and software latencies.

In the light bulb and sunrise example in the previous section, if we choose a granularity of 1 second, then the switching on of a light bulb is an event that occurs at a specific time point. A granularity of 1 second, however, means that a sunrise is not an event but an interval, given the fact that in most parts of the world, the sun takes a few minutes to rise above the horizon. However, if we choose a granularity of 1 hour (and we do not live in the polar regions), a sunrise becomes an event that occurs at a single point in our chosen time scale. We can therefore argue that given a specific time granularity, some events can be treated as instantaneously occurring at a specific time point.

Regardless of the granularity chosen, the domain of time points is the set of positive integers $\mathbb{Z}^+$.

4.3.2 Defining Intervals in Terms of Time Points

Definition 4.3 *A* **well-defined interval** ι *is an interval whose end points* $\iota^- \in \mathbb{Z}^+$ *and* $\iota^+ \in \mathbb{Z}^+$ *are defined, where* $\iota^- < \iota^+$. *A* **regular interval** *is an interval whose end points are not known.*

Under this definition, we can conclude that for any two well-defined intervals there is exactly one interval relation that holds between them. Algorithm 4.3 shows a function that calculates this relation given the end points of two well-defined intervals.

By using the $Compute()$ function in Algorithm 4.3, we can now define a network operator, $NET.Bind(\iota, \iota^-, \iota^+)$, that assigns the end points ι^- and ι^+ to the existing interval ι in the network. By allowing such end points to be defined for any interval, thereby making them well-defined intervals, we are also allowing the possibility of introducing new relations with this interval. Furthermore, any new relations gathered by comparing the end points of well-defined intervals are subject to the same propagation algorithms shown in the previous section. Algorithm 4.4 shows how the $NET.Bind()$ operator achieves this.

Algorithm 4.3 *Compute()*

```
FUNCTION Compute(ι0⁻, ι0⁺, ι1⁻, ι1⁺)
  IF ι0⁻ == ι1⁻ THEN
    IF ι0⁺ == ι1⁺ THEN
      RETURN {equals}
    ELSE IF ι0⁺ < ι1⁺ THEN
      RETURN {starts}
    ELSE
      RETURN {started by}
```

```
    ENDIF
  ELSE IF ι_0^- < ι_1^- THEN
    IF ι_0^+ == ι_1^+ THEN
      RETURN {finished by}
    ELSE IF ι_0^+ < ι_1^+ THEN
      IF ι_0^+ == ι_1^- THEN
        RETURN {meets}
      ELSE IF ι_0^+ < ι_1^- THEN
        RETURN {before}
      ELSE
        RETURN {overlaps}
      ENDIF
    ELSE
      RETURN {contains}
    ENDIF
  ELSE
    IF ι_0^+ == ι_1^+ THEN
      RETURN {finishes}
    ELSE IF ι_0^+ < ι_1^+ THEN
      RETURN {during}
    ELSE
      IF ι_0^- == ι_1^+ THEN
        RETURN {met by}
      ELSE IF ι_0^- < ι_1^+ THEN
        RETURN {overlapped by}
      ELSE
        RETURN {after}
      ENDIF
    ENDIF
  ENDIF
ENDFUNCTION
```

Algorithm 4.4 *Net.Bind()*

```
FUNCTION NET.Bind(ι, ι^-, ι^+)
  FOR each interval ι' ∈ NET DO
    IF ι' != ι THEN
      IF ι' is well-defined THEN
        rs = Compute(ι^-, ι^+, ι'^-, ι'^+)
        NET.AddRel(ι, ι', rs)
      ENDIF
    ENDIF
  ENDDO
```

```
ENDFUNCTION
```

4.4 Formalisation

In this section, we attempt to formalise a new language, $\mathcal{L}^{\mathcal{T}}$, with the same expressive power as language $\mathcal{L}$ to represent authorisation policies, but with extensions to also express temporal constraints. Like language $\mathcal{L}$, language $\mathcal{L}^{\mathcal{T}}$ is also a first-order logic language. As such, any language $\mathcal{L}^{\mathcal{T}}$ policy can be translated into a normal logic program to derive answer sets, from which queries can be evaluated. Section A.2 contains the full BNF specification of language $\mathcal{L}^{\mathcal{T}}$.

4.4.1 Syntax

Components of Language $\mathcal{L}^{\mathcal{T}}$

- **Identifiers**

  In language $\mathcal{L}^{\mathcal{T}}$, there are 4 general types of identifiers:

  1. *Entity Identifiers.* As with language $\mathcal{L}$, language $\mathcal{L}^{\mathcal{T}}$ includes six disjoint entity sorts: subject, access rights, objects, subject groups, access right groups and object groups. The syntax for each entity type is a single lower case alpha character followed by zero or more alphanumeric or underscore characters:

     ```
     [a-z][a-zA-Z0-9_]
     ```

  2. *Interval Identifiers.* The main difference between language $\mathcal{L}$ and language $\mathcal{L}^{\mathcal{T}}$ is that in addition to the six entity sorts, language $\mathcal{L}^{\mathcal{T}}$ also includes an additional time interval sort. As it occupies a different name space from the other sorts, interval identifiers share the same syntax.

  3. *Policy Update Identifiers.* These identifiers are used as labels to name policy updates. They occupy a different name space from other identifiers and hence share the same syntax as entity and interval identifiers.

  4. *Variable Identifiers.* In language $\mathcal{L}$, variable identifiers are used to represent entity identifiers. In contrast, language $\mathcal{L}^{\mathcal{T}}$ variable identifiers are used to represent both entity and interval identifiers. The list below shows the syntax of the different types of variable identifiers:

     (a) Subject Variables

     ```
     S[SG][a-zA-Z0-9_]
     ```

(b) Access Right Variables

`A[SG][a-zA-Z0-9_]`

(c) Object Variables

`O[SG][a-zA-Z0-9_]`

(d) Interval Variables

`I[a-zA-Z0-9_]`

- **Authorisation Atoms, Facts and Expressions**

  Authorisation atoms of language $\mathcal{L}^{\mathcal{T}}$ are similar to the to the atoms of language $\mathcal{L}$, except each atom includes an interval parameter that indicates the time at which that atom holds. As with language $\mathcal{L}$, there are three types of atoms:

  1. *Holds Atom.* The syntax below shows an atom that states that subject $sub\text{-}id$ holds the access right $acc\text{-}id$ for object $obj\text{-}id$ at time interval $int\text{-}id$.

     ```
     holds(<sub-id>,
           <acc-id>,
           <obj-id>,
           <int-id>)
     ```

  2. *Member Atom.* The syntax shown below is that of an atom that states that the single entity $single\text{-}id$ is a member of the group entity $group\text{-}id$ for the duration specified by interval $int\text{-}id$.

     ```
     memb(<single-id>, <group-id>, <int-id>)
     ```

  3. *Subset Atom.* Below is the syntax of an atom that states that the group entity $group\text{-}id\text{-}0$ is a subset of the group entity $group\text{-}id\text{-}1$ at time interval $int\text{-}id$.

     ```
     subst(<group-id-0>, <group-id-1>, <int-id>)
     ```

  Like language $\mathcal{L}$, language $\mathcal{L}^{\mathcal{T}}$ *facts* state that relationship represented by the corresponding atom its negation (as indicated by the "!" character prefix) holds.

  An *expression* is either a fact or a conjunction of facts separated by the comma "," character. An atom, fact or expression composed entirely of entity and interval identifiers (no variables) are called *ground atoms*, *facts* or *expressions*, respectively.

- **Interval Atoms and Expressions**

Noting the fact that the inverse relation between any intervals i_0 and i_1 is equal to the relation between intervals i_1 and i_0, for the sake of brevity, language $\mathcal{L}^{\mathcal{T}}$ defines only 7 out of the 13 interval relations in the algebra. These relations, together with two interval identifiers make up the language's interval atoms:

```
<rel-id>(<int-id-0>, <int-id-1>)
```

where $rel\text{-}id$ indicates the relation between interval $int\text{-}id$-0 and interval $int\text{-}id$-1, and is one of the following: $equals$, $before$, $during$, $overlaps$, $meets$, $starts$ or $finishes$.

An interval expression in language $\mathcal{L}^{\mathcal{T}}$ is a group of interval atoms separated by the comma "," character. As the interval algebra allows disjunctions, the meaning of the comma within an interval expression may be conjunctive or disjunctive. If an interval expression contains two atoms that differ only by their relation (meaning the interval pairs are the same), then the comma between these atoms indicates a logical *or*. Commas between other interval atoms indicate a logical *and*. For example, the following interval expression is interpreted as "interval i_0 is before or after interval i_1 and interval i_2 is during interval i_0":

```
before(i0, i1), before(i1, i0), during(i2, i0)
```

Like their authorisation counterparts, interval atoms and expressions that do not have variable identifiers are called ground interval atoms and expressions.

Identifier Declarations

In language $\mathcal{L}^{\mathcal{T}}$, both entity and interval identifiers must be declared before they are used anywhere in the program. The syntax for declaring entity identifiers in language $\mathcal{L}^{\mathcal{T}}$ is the same as the syntax in language $\mathcal{L}$:

```
entity sub|acc|obj[-grp] <ent-id>[, ...];
```

Interval identifiers may be declared with or without end points. Note that once intervals are assigned end points, they are bound to those end points for the lifetime of the program. End points, if declared with an interval identifier, must be a positive integer as indicated in the following syntax:

```
interval <int-id> [\[ep0, ep1\]][, ... ];
```

where $ep_0 \in \mathbb{Z}^+$, $ep_1 \in \mathbb{Z}^+$ and $ep_0 < ep_1$.

Initial Fact Declarations

Like language $\mathcal{L}$, language $\mathcal{L}^{\mathcal{T}}$ allows the declaration of initial state facts. Ground facts declared in this manner hold until a policy update causes them to be otherwise. The syntax shown below declares all facts in the ground authorisation expression $gnd\text{-}auth\text{-}exp$ as initial state facts.

```
initially <gnd-auth-exp>;
```

Interval Constraint Declarations

Relations between intervals are expressed in language $\mathcal{L}^{\mathcal{T}}$ through interval constraints. Interval relations defined by interval constraints hold for the entire lifetime of the program. Such relations may be declared in the following manner:

```
relation <gnd-int-exp>;
```

where $gnd\text{-}int\text{-}exp$ is a ground interval expression.

The example below declares an interval constraint that states that interval i_0 is $before$ or $starts$ interval i_1, interval i_1 is $during$ or $meets$ interval i_2, and interval i_2 $meets$ interval i_3:

```
relation
  before(i0, i1),
  starts(i0, i1),
  during(i1, i2),
  meets(i1, i2),
  meets(i2, i3);
```

Authorisation Constraint Declarations

Like the constraint declarations in language $\mathcal{L}$, authorisation constraint declarations in language $\mathcal{L}^{\mathcal{T}}$ are used to define logical rules that always holds, even after a policy update is applied. The difference, as shown by the syntax below, is the addition of the *where* clause.

```
always <auth-exp-0>
  [implied by <auth-exp-1>]
  [with absence <auth-exp-2>]
  [where <int-exp>];
```

The meaning of the statement is essentially the same as its language $\mathcal{L}$ counterpart: expression $auth\text{-}exp$-0 holds if expression $auth\text{-}exp$-1 holds and there is no evidence that expression $auth\text{-}exp$-2 holds. Any variables occurring in any of these expressions are grounded to all defined entities and interval identifiers.

The *where* clause is used to define an interval expression $int\text{-}exp$ which is used to place a restriction on the interval identifiers used to ground interval variables occurring in the authorisation expressions. Only those sets of interval identifiers that satisfy the expression $int\text{-}exp$ is used to replace the set of interval variables in the authorisation expressions. As the *where* clause is used in the grounding of variables, it is important to note that it does not make sense to have a ground interval expression in the *where* clause. Furthermore, not only does the expression $int\text{-}exp$ need to be non-ground, but every atom in this expression must also be non-ground.

For example, given the following authorisation constraint declaration:

```
always holds(SS, a, o, I0)
  implied by memb(SS, SG0, I1)
  with absence !memb(SS, SG1, I1)
  where starts(I0, I1);
```

Assuming that $\{ss_0, ss_1\}$ is the set of single subjects, $\{sg_0, sg_1\}$ is the set of group subjects and $\{i_0, i_1, i_2\}$ is the set of intervals defined, with interval i_0 $during$ interval i_1 and interval i_1 $start$ing interval i_2, the statement is equivalent to the following statements:

```
always holds(ss0, a, o, i1)
  implied by memb(ss0, sg0, i2)
  with absence !memb(ss0, sg1, i2);

always holds(ss1, a, o, i1)
  implied by memb(ss1, sg0, i2)
  with absence !memb(ss1, sg1, i2);

always holds(ss0, a, o, i1)
  implied by memb(ss0, sg1, i2)
  with absence !memb(ss0, sg0, i2);

always holds(ss1, a, o, i1)
  implied by memb(ss1, sg1, i2)
  with absence !memb(ss1, sg0, i2);
```

Although the entity variables SS, SG_0 and SG_1 are grounded to every entity that matches their respective types, the interval variables I_0 and I_1 are restricted to the intervals i_1 and i_2, respectively, because they are the only interval pair that satisfies the restriction placed by the *where* clause.

Policy Update Declarations

Like the authorisation constraint definition, policy update definitions of language $\mathcal{L}^{\mathcal{T}}$ are similar to those of language $\mathcal{L}$, but with the extra *where* clause to limit the interval identifiers that are used to ground interval variables that may occur in authorisation expressions.

The syntax below describes the declaration of a policy update $update\text{-}id$ which, when applied, causes the expression $auth\text{-}exp$-0 to hold if the expression $auth\text{-}exp$-1 already holds.

```
<update-id>([<ent-var-0>[, ...]])
  causes <auth-exp-0>
  [if <auth-exp-1>]
  [where <int-exp>];
```

When a policy update is applied, entity identifiers are supplied for each entity variable $ent\text{-}var\text{-}i$. These entity identifiers are used to ground any matching entity variables that may occur in either authorisation expressions $auth\text{-}exp$-0 or $auth\text{-}exp$-1. Entity variables occurring in these expressions that do not match the variables in the variable list $ent\text{-}var\text{-}n$ are replaced by all defined entity identifiers that match the variable types. Interval variables occurring in the authorisation expressions are grounded to sets of interval identifiers that satisfy the interval expression $int\text{-}exp$.

Policy Update Directives

The policy update directives in language $\mathcal{L}^{\mathcal{T}}$, like those in language $\mathcal{L}$, are used to manipulate the policy update sequence list.

The three directives below are for adding a policy update into the update sequence list, removing an update from the update sequence list, and showing the contents of the update sequence list, respectively.

```
seq add <upd-id>([<id-0>[, ...]]);
seq del <n>;
seq list;
```

The directive is used to apply policy updates one at a time in the order in which they appear in the update sequence list.

```
compute;
```

Query Directives

A ground query expression may be given to the system for evaluation. The syntax is as follows:

```
query <gnd-auth-exp>;
```

Queries are evaluated against the policy base state derived from the application of policy updates in the update sequence list. The system response for each query directive either *true*, *false* or *unknown*.

Example 4.2 *The example code below shows a full language $\mathcal{L}^{\mathcal{T}}$ policy description. In this policy, three intervals are defined: $work\_hours$, $morning\_hours$ and $afternoon\_hours$, where $worh\_hours$ starts at 9:00 AM and ends at 5:00 PM. Furthermore, the interval constraint in the policy states that $morning\_hours$ either* starts *or is* during *$work\_hours$, $afternoon\_hours$ either* finishes *or is* during *$work\_hours$, and $morning\_hours$ is* before *$afternoon\_hours$.*

The authorisation constraint states that for all defined intervals $I0$, if $grp1$ holds the $read$ access right to $file$ at interval $I0$, and there is no evidence that $grp3$ does not hold the $write$ access right to $file$ at interval $I0$, then $grp1$ holds the $write$ access right to $file$ at interval $I0$.

The policy update $delete\_read$ definition states that when applied, the update will cause some subject group $SG0$ to lose the $read$ access right to some object $OS0$ at all intervals $I0$ that either starts *or is* during *the interval $work\_hours$.*

```
/* entity declarations */

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

/* interval declarations */
```

```
interval work_hours [0900, 1700];
interval morning_hours;
interval afternoon_hours;

/* initial fact statement */

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

/* interval constraints */

relation
  during(morning_hours, work_hours),
  starts(morning_hours, work_hours);

relation
  during(afternoon_hours, work_hours),
  finishes(afternoon_hours, work_hours);

relation
  before(morning_hours, afternoon_hours);

/* authorisation constraint */

always holds(grp1, write, file, I0)
  implied by
    holds(grp1, read, file, I0)
  with absence
    !holds(grp3, write, file, I0);

/* policy update declaration */

delete_read(SG0, OS0)
  causes
    !holds(SG0, read, OS0, I0)
```

```
  where
    starts(work_hours, I0),
    during(work_hours, I0);

/* add delete_read to policy update sequence list */

seq add delete_read(grp1, file);

compute;

/* queries */

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

4.4.2 Semantics

The domain description $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ of language $\mathcal{L}^{\mathcal{T}}$ is a finite set of intervals with end points, initial state facts, temporal constraint rules, authorisation constraint rules, policy update definitions and an ordered set ψ of policy update references.

Like language $\mathcal{L}$, the semantics of language $\mathcal{L}^{\mathcal{T}}$ is best described by its translation into an extended logic program, language $\mathcal{L}^{\mathcal{T}^*}$. Formally, given a domain description $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ of language $\mathcal{L}^{\mathcal{T}}$, the translation is denoted by $Trans(\mathcal{D}_{\mathcal{L}^{\mathcal{T}}})$.

The main difference between language $\mathcal{L}^{\mathcal{T}}$ and language $\mathcal{L}^{\mathcal{T}^*}$ is that each atom of the latter also specifies the policy update state in which it holds. Another difference between the two languages is that the temporal constraints of language $\mathcal{L}^{\mathcal{T}}$ is not directly expressed in language $\mathcal{L}^{\mathcal{T}^*}$. Instead, the interval algebra discussed at the beginning of this chapter is used by the translation process to generate the appropriate authorisation rules with respect to the given temporal constraints.

Before the translation process can be shown, we must first provide a formal definition of language $\mathcal{L}^{\mathcal{T}^*}$.

Language $\mathcal{L}^{\mathcal{T}^*}$

As hinted above, language $\mathcal{L}^{\mathcal{T}^*}$ is an extended logic program which is composed of facts and rules that expresses an authorisation policy. The following are the components of the

language:

- **Entities**

  Like language $\mathcal{L}^*$, language $\mathcal{L}^{\mathcal{T}^*}$ defines a set $\mathcal{E}$ that contains all the defined subject, access right and object (both singles and groups) entities. In addition to this set, we also define all its subsets: single subject $\mathcal{E}_{ss}$, single access right $\mathcal{E}_{as}$, single object $\mathcal{E}_{os}$, group subject $\mathcal{E}_{sg}$, group access right $\mathcal{E}_{ag}$, group object $\mathcal{E}_{og}$, single and group subjects $\mathcal{E}_s$, single and group access rights $\mathcal{E}_a$, and single and group objects $\mathcal{E}_o$.

- **Intervals**

  Language $\mathcal{L}^{\mathcal{T}^*}$ defines a set $\mathcal{I}$ that contains all the defined time intervals from $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$.

- **Atoms**

  A language $\mathcal{L}^{\mathcal{T}^*}$ atom is a binding of a set of entities, a temporal interval and a state. The set $\mathcal{A}^\sigma$ contains all the atoms of state σ. The sets $\mathcal{A}^\sigma_h$, $\mathcal{A}^\sigma_m$ and $\mathcal{A}^\sigma_h$ denotes all $holds$, $member$ and $subset$ atoms of state σ, respectively, where $\mathcal{A}^\sigma = \mathcal{A}^\sigma_h \cup \mathcal{A}^\sigma_m \cup \mathcal{A}^\sigma_s$. The definitions of these subsets are shown below. To distinguish between the atoms of the two languages, atoms of language $\mathcal{L}^{\mathcal{T}^*}$ are written with the hat character.

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

- **Facts**

  The definition below states that a fact $\hat{\rho}^\sigma$ is a logical statement that asserts that an atom $\hat{\alpha}$ either holds or does not hold at a given state σ.

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

- **Expressions**

  A language $\mathcal{L}^{\mathcal{T}^*}$ expression is a conjunction of facts separated by the comma character. The expression below asserts that where $0 \leq i \leq n$, each fact $\hat{\rho}_i$ holds:

  $\hat{\rho}_0, \hat{\rho}_1, \ldots, \hat{\rho}_n,$

Translating Language $\mathcal{L}^{\mathcal{T}}$ to Language $\mathcal{L}^{\mathcal{T}^*}$

Unlike the translation of language $\mathcal{L}$ to language $\mathcal{L}^*$, not all statements of language $\mathcal{L}^{\mathcal{T}}$ is translated to language $\mathcal{L}^{\mathcal{T}^*}$. In particular, as language $\mathcal{L}^{\mathcal{T}^*}$ does not express relationships between temporal intervals, all language $\mathcal{L}^{\mathcal{T}}$ statements that denote these relationships are not directly translated. Through the use of Allen's interval algebra, the authorisation rules that result from the consequences of these temporal relations are generated instead.

To describe the details of the translation $Trans(\mathcal{D}_{\mathcal{L}^{\mathcal{T}}})$, we first define some translation functions similar to those defined in Section 2.2.2 and Section 2.2.3.

- The $Res(u, \sigma)$ function takes as input a policy update $u \in \psi$ and a policy state σ, and returns the policy state that results from applying the update u upon state σ.
- The $CopyAtom(\hat{\alpha}, \sigma)$ function takes as input an atom $\hat{\alpha}$ of language $\mathcal{L}^{\mathcal{T}^*}$ and some state σ, then returns an atom with the same type and with the same entities and interval as atom $\hat{\alpha}$, but with state σ instead of the original state specified by $\hat{\alpha}$.
- The $TransAtom(\alpha, \sigma)$ function takes as input an atom α of language $\mathcal{L}^{\mathcal{T}}$ and some state σ, then returns an equivalent atom of language $\mathcal{L}^{\mathcal{T}^*}$ with the same entities and interval specified by α.
- The $TransFact(\rho, \sigma)$ function is similar to the $TransAtom$ function, but instead of translating an atom, it takes a language $\mathcal{L}^{\mathcal{T}}$ fact ρ and some state σ then returns the equivalent language $\mathcal{L}^{\mathcal{T}^*}$ fact.

With these functions defined, we can now outline the translation process:

Initialising the Temporal Interval Relation Network. The first step in the process is to initialise the interval relation network with all the temporal intervals and all the given end points defined in language $\mathcal{L}^{\mathcal{T}}$. Recall that there are two ways a temporal interval may be declared in language $\mathcal{L}^{\mathcal{T}}$:

```
interval ι;
interval ι [ep0, ep1];
```

For both forms, the interval ι is added to the interval network. However, the second form requires an additional step to register the end points:

$NET.Bind(\iota, ep_0, ep_1)$

Populating the Temporal Interval Relation Network. The next step is to encode all temporal constraints in language $\mathcal{L}^{\mathcal{T}}$ as interval relations in the interval relation network. Temporal constraints are declared in language $\mathcal{L}^{\mathcal{T}}$ in the following way:

`relation` α_0`, ...,` α_n`;`

Note that each interval atom α_i above is in the form $r_i(\iota 0_i, \iota 1_i)$, where $r_i \in \{equals, before, \ldots\}$, $\iota 0_i, \iota 1_i \in \mathcal{I}$ and $0 \leq i \leq n$. We further note that under this notation, it is possible to encounter a situation where two different interval atoms α_i and α_j can both contain the same pair of intervals:

$\alpha_i = r_i(\iota 0_i, \iota 1_i)$

$\alpha_j = r_j(\iota 0_j, \iota 1_j)$

where

$i \neq j$,

$r_i \neq r_j$,

$\iota 0_i = \iota 0_j$,

$\iota 1_i = \iota 1_j$

According to the syntax definition of interval atoms in the previous section, in any given interval expression, all interval atom pairs that satisfies the above condition are to be treated as disjunctions in that expression, while those that do not are to be treated as conjunctions.

For simplicity, we introduce a normalised notation for interval expressions where each interval atom in the expression contains a unique pair of intervals and a set of relations that hold between those intervals. Formally, a normalised interval expression is in the following form:

$\alpha_0, \ldots, \alpha_n$

where each interval atom α_i $(0 \leq i \leq n)$ in the form:

$\alpha_i = (\iota 0_i, \iota 1_i, \{r0_i, \ldots, rx_i\})$

satisfies the following condition:

$\neg\exists\alpha_j$,

where

α_j = ($\iota 0_j$, $\iota 1_j$, $\{r0_j, \ldots, ry_j\}$)

$0 \leq j \leq n$,

$i \neq j$

such that

$\iota 0_i = \iota 0_j$,

$\iota 1_i = \iota 1_j$

Now, we define a function $NormaliseExp(\epsilon)$ that takes a language $\mathcal{L}^{\mathcal{T}}$ interval expression ϵ as input and returns the normalised equivalent of that expression in the form shown above. Formally, we have:

$\epsilon' = NormaliseExp(\epsilon)$

For each ($\iota 0_i$, $\iota 1_i$, $\{r0_i, \ldots, rx_i\}$) $\in \epsilon'$, where $0 \leq i < |\epsilon'|$, the following conditions are satisfied:

1. For each $r_j \in \{r0_i, \ldots, rx_i\}$, where $0 \leq j \leq i$, there exists an $r_j(\iota 0_i, \iota 1_i) \in \epsilon$
2. For each ($\iota 0_j$, $\iota 1_j$, $\{r0_j, \ldots, ry_j\}$) $\in \epsilon'$, where $0 \leq j < |\epsilon'|$ and $i \neq j$, ($\iota 0_i \neq \iota 0_j \vee \iota 1_i \neq \iota 1_j$)

With the $NormaliseExp()$ function defined, we can now populate the interval relation network with relations expressed in language $\mathcal{L}^{\mathcal{T}}$ interval constraint statements. The steps involved to achieve this is as follows. For each interval constraint statement in language $\mathcal{L}^{\mathcal{T}}$:

`relation` α_0`, ..., `α_n`;`

By using the $NormaliseExp()$ function, we obtain the normalised expression ϵ:

$\epsilon = NormaliseExp(\{\alpha_0, \ldots, \alpha_n\})$

Finally, to register the interval constraint to the interval relation network, we make a call to the following operator for each ($\iota 0_i$, $\iota 1_i$, rs_i) $\in \epsilon$, where $0 \leq i < |\epsilon|$:

$NET.AddRel$`(`$\iota 0_i$`, `$\iota 1_i$`, `rs_i`)`

Variable Grounding. As language $\mathcal{L}^{\mathcal{T}^*}$ does not allow variables, all language $\mathcal{L}^{\mathcal{T}}$ expressions containing entity or interval variables must be grounded in the translation process. Although the task of grounding entity variables is a relatively straightforward procedure, grounding interval variables involves additional steps due to the *where* clause of the language. After generating the tuples to replace the variables, each of these tuples must also be checked to ensure that they satisfy any given temporal constraints.

Before we can describe the grounding process in greater detail, we first define the following three functions:

- $Type(ev)$

  The $Type(ev)$ function returns the type of the given entity or variable ev: $interval$, $single\text{-}subject$, $group\text{-}subject$, $single\text{-}access\text{-}right$, etc.

- $Var(\epsilon)$

  Given a non-ground language $\mathcal{L}^{\mathcal{T}}$ authorisation or interval expression ϵ, this function returns a set of unique variables that occurs in ϵ. If ϵ is ground, the function returns $\emptyset$.

- $Replace(\epsilon, V, t)$

  The function takes the following as input: a non-ground language $\mathcal{L}^{\mathcal{T}}$ authorisation or interval expression ϵ, a set of variables V that occur in ϵ, and a tuple t containing entities and intervals that correspond to the variables in V ($|V| = |t|$ and $\forall i$, $0 \leq i < |V|$, $Type(V_i) = Type(t_i)$). The function returns ϵ with all variable occurences replaced with the corresponding entities or intervals from t. If ϵ is a ground expression, ϵ is returned.

With these functions defined, we can now generalise the process of grounding both entity and interval variables. Given a set V of unique entity and interval variables, the function below returns a set of all possible $|V|$-tuples that can be used to replace the variables in V:

$$GenTuples_1(V) = R_0 \times \ldots \times R_{|V|-1}$$

where

$$\forall\, i, 0 \leq i < |V|,$$

$$R_i = \begin{cases} \mathcal{E}_s, \text{ if } Type(V_i) = \textit{subject} \\ \mathcal{E}_a, \text{ if } Type(V_i) = \textit{access right} \\ \vdots \\ \mathcal{I}, \text{ if } Type(V_i) = \textit{interval} \end{cases}$$

A variation of this function, shown below, generates tuples with respect to a given temporal constraint. Given a language $\mathcal{L}^{\mathcal{T}}$ *where* clause with an interval expression ϵ and a set V of variables that occur in ϵ, the following function returns a set of $|V|$-tuples, where each tuple satisfies the condition given in ϵ:

$$GenTuples_2(\epsilon, V) = \{t_0, \ldots, t_{|V|-1}\}$$

where

$$\forall\, i, 0 \leq i < |V|, t_i \in GenTuples_1(V),$$
$$\epsilon' = NormaliseExp(Replace(\epsilon, V, t_i)),$$
$$\forall\, (\iota_0, \iota_1, rs) \in \epsilon', NET.Get(\iota_0, \iota_1) \subseteq rs$$

Initial Fact Rules. The initial fact rules are obtained directly from initial fact declaration statements of language $\mathcal{L}^{\mathcal{T}}$:

`initially` ρ_0`,` $\ldots$`,` ρ_n`;`

Each initial fact declaration statement of language $\mathcal{L}^{\mathcal{T}}$ in the form above corresponds to the following language $\mathcal{L}^{\mathcal{T}^*}$ rules:

$$\hat{\rho}_i \leftarrow$$

where

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

Authorisation Constraint Rules. An authorisation constraint statement in language $\mathcal{L}^{\mathcal{T}}$ is in the form:

`always` ρ_{0_0}`,` $\ldots$`,` $\rho_{0_{n_0}}$
`implied by` ρ_{1_0}`,` $\ldots$`,` $\rho_{1_{n_1}}$
`with absence` ρ_{2_0}`,` $\ldots$`,` $\rho_{2_{n_2}}$
`where` r_0`,` $\ldots$`,` r_{n_3}`;`

The first step is to gather all the entity and interval variables that occur in all the expressions above into a set V:

$$V = Var((\rho_{0_0}, \ldots, \rho_{0_{n_0}})) \cup Var((\rho_{1_0}, \ldots, \rho_{1_{n_1}})) \cup Var((\rho_{2_0}, \ldots, \rho_{2_{n_2}}))$$

Using the set V, we generate a set of tuples T that satisfies the temporal constraint specified by the *where* clause:

$$T = GenTuples_2((r_0, \ldots, r_{n_3}), V)$$

Finally, the language $\mathcal{L}^{\mathcal{T}^*}$ equivalent of the authorisation constraint rule is the following rules:

$\forall\ (t, \sigma)$,

$$\hat{\rho}^{\sigma}_{0_0} \leftarrow \hat{\rho}^{\sigma}_{1_0}, \ldots, \hat{\rho}^{\sigma}_{1_{n_1}}, not\ \hat{\rho}^{\sigma}_{2_0}, \ldots, not\ \hat{\rho}^{\sigma}_{2_{n_2}}$$

$$\vdots$$

$$\hat{\rho}^{\sigma}_{0_{n_0}} \leftarrow \hat{\rho}^{\sigma}_{1_0}, \ldots, \hat{\rho}^{\sigma}_{1_{n_1}}, not\ \hat{\rho}^{\sigma}_{2_0}, \ldots, not\ \hat{\rho}^{\sigma}_{2_{n_2}}$$

where

$\hat{\rho}^{\sigma}_{0_i} = TransFact(Replace(\rho_{0_i}, V, t), \sigma), 0 \leq i \leq n_0,$

$\hat{\rho}^{\sigma}_{1_j} = TransFact(Replace(\rho_{1_j}, V, t), \sigma), 0 \leq j \leq n_1,$

$\hat{\rho}^{\sigma}_{2_k} = TransFact(Replace(\rho_{1_k}, V, t), \sigma), 0 \leq k \leq n_2,$

$t \in T,$

$S_0 \leq \sigma \leq S_{|\psi|}$

Policy Update Rules. Obviously, only language $\mathcal{L}^{\mathcal{T}}$ policy update statements that are applied to the policy are actually translated to language $\mathcal{L}^{\mathcal{T}^*}$ rules. The translation process for these rules are again similar to that of language $\mathcal{L}^*$, except this time, the variable grounding is subject to the constraints specified by the *where* clause. A language $\mathcal{L}^{\mathcal{T}}$ policy update statement is shown below:

u `causes` ρ_{0_0}`, ...,` $\rho_{0_{n_0}}$
`if` ρ_{1_0}`, ...,` $\rho_{1_{n_1}}$
`where` r_0`, ...,` r_{n_2}`;`

Like the authorisation constraint rules, the first step in the translation process is to generate a set V of entity and interval variables that occur in all the expressions:

$$V = Var(\rho_{0_0}, \ldots, \rho_{0_{n_0}}) \cup Var(\rho_{1_0}, \ldots, \rho_{1_{n_1}})$$

With the variable set V and a *where* clause expression $(r_0, \ldots, r_{n_2})$, a set of tuples T can be generated such that each tuple $t \in T$ satisfies the *where* clause constraint:

$T = GenTuples_2((r_0, \ldots, r_{n_2}), V)$

By using the entities and intervals in each tuple in set T to ground all the variables, we can now define the policy update rules in language $\mathcal{L}^{\mathcal{T}^*}$:

$\forall\, t,$

$\hat{\rho}_{0_0} \leftarrow \hat{\rho}_{1_0}, \ldots, \hat{\rho}_{1_{n_1}}$

$\vdots$

$\hat{\rho}_{0_{n_0}} \leftarrow \hat{\rho}_{1_0}, \ldots, \hat{\rho}_{1_{n_1}}$

where

$\hat{\rho}_{0_i} = TransFact(Replace(\rho_{0_i}, V, t), Res(u, \sigma)), 0 \leq i \leq n_0,$

$\hat{\rho}_{1_j} = TransFact(Replace(\rho_{1_j}, V, t), \sigma), 0 \leq j \leq n_1,$

$t \in T,$

$S_0 \leq \sigma \leq S_{|\psi|}$

Inheritance Rules. As with language $\mathcal{L}^*$, a set of language $\mathcal{L}^{\mathcal{T}^*}$ rules are needed to express the inheritance properties of members and subsets. While these rules are similar to their respective language $\mathcal{L}^*$ counterparts, the definitions are slightly different due to the representation of temporal intervals.

1. Subject Group Member Inheritance Rules

   $\forall\, (ss, sg, a, o, \iota\, \sigma),$

   $\hat{holds}(ss, a, o, \iota, \sigma) \leftarrow$
   $\quad \hat{holds}(sg, a, o, \iota, \sigma), \hat{memb}(ss, sg, \iota, \sigma),$
   $\quad not\ \neg\hat{holds}(ss, a, o, \iota, \sigma)$

   $\neg\hat{holds}(ss, a, o, \iota, \sigma) \leftarrow$
   $\quad \neg\hat{holds}(sg, a, o, \iota, \sigma), \hat{memb}(ss, sg, \iota, \sigma)$

   where
   $\quad ss \in \mathcal{E}_{ss},$
   $\quad sg \in \mathcal{E}_{sg},$
   $\quad a \in \mathcal{E}_a,$

$o \in \mathcal{E}_o$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

2. Access Right Group Member Inheritance Rules

$\forall\ (s, as, ag, o, \iota\ \sigma)$,

$\hat{holds}(s, as, o, \iota, \sigma) \leftarrow$
$\hat{holds}(s, ag, o, \iota, \sigma), \hat{memb}(as, ag, \iota, \sigma),$
$not\ \neg\hat{holds}(s, as, o, \iota, \sigma)$

$\neg\hat{holds}(s, as, o, \iota, \sigma) \leftarrow$
$\neg\hat{holds}(s, ag, o, \iota, \sigma), \hat{memb}(as, ag, \iota, \sigma)$

where
$s \in \mathcal{E}_s$,
$as \in \mathcal{E}_{as}$,
$ag \in \mathcal{E}_{ag}$,
$o \in \mathcal{E}_o$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

3. Object Group Member Inheritance Rules

$\forall\ (s, a, os, og, \iota\ \sigma)$,

$\hat{holds}(s, a, os, \iota, \sigma) \leftarrow$
$\hat{holds}(s, a, og, \iota, \sigma), \hat{memb}(os, og, \iota, \sigma),$
$not\ \neg\hat{holds}(s, a, os, \iota, \sigma)$

$\neg\hat{holds}(s, a, os, \iota, \sigma) \leftarrow$
$\neg\hat{holds}(s, a, og, \iota, \sigma), \hat{memb}(os, og, \iota, \sigma)$

where
$s \in \mathcal{E}_s$,
$a \in \mathcal{E}_a$,
$os \in \mathcal{E}_{os}$,

$og \in \mathcal{E}_{og}$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

4. Subject Group Subset Inheritance Rules

$\forall\ (sg_0, sg_1, a, o, \iota, \sigma)$,

$\hat{holds}(sg_0, a, o, \iota, \sigma) \leftarrow$
$\hat{holds}(sg_1, a, o, \iota, \sigma), \hat{subst}(sg_0, sg_1, \iota, \sigma)$,
$not\ \neg\hat{holds}(sg_0, a, o, \iota, \sigma)$

$\neg\hat{holds}(sg_0, a, o, \iota, \sigma) \leftarrow$
$\neg\hat{holds}(sg_1, a, o, \iota, \sigma), \hat{subst}(sg_0, sg_1, \iota, \sigma)$

where
$sg_0, sg_1 \in \mathcal{E}_{sg}$,
$a \in \mathcal{E}_a$,
$o \in \mathcal{E}_o$,
$sg_0 \neq sg_1$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

5. Access Right Group Subset Inheritance Rules

$\forall\ (s, ag_0, ag_1, o, \iota, \sigma)$,

$\hat{holds}(s, ag_0, o, \iota, \sigma) \leftarrow$
$\hat{holds}(s, ag_1, o, \iota, \sigma), \hat{subst}(ag_0, ag_1, \iota, \sigma)$,
$not\ \neg\ \hat{holds}(s, ag_0, o, \iota, \sigma)$

$\neg\hat{holds}(s, ag_0, o, \iota, \sigma) \leftarrow$
$\neg\hat{holds}(s, ag_1, o, \iota, \sigma), \hat{subst}(ag_0, ag_1, \iota, \sigma)$

where
$s \in \mathcal{E}_s$,
$ag_0, ag_1 \in \mathcal{E}_{ag}$,
$o \in \mathcal{E}_o$,

$ag_0 \neq ag_1$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

6. Object Group Subset Inheritance Rules

$\forall\ (s, a, og_0, og_1, \iota, \sigma)$,

$\hat{holds}(s, a, og_0, \iota, \sigma) \leftarrow$
$\hat{holds}(s, a, og_1, \iota, \sigma), \hat{subst}(og_0, og_1, \iota, \sigma)$,
$not\ \neg\hat{holds}(s, a, og_0, \iota, \sigma)$

$\neg\hat{holds}(s, a, og_0, \iota, \sigma) \leftarrow$
$\neg\hat{holds}(s, a, og_1, \iota, \sigma), \hat{subst}(og_0, og_1, \iota, \sigma)$

where
$s \in \mathcal{E}_s$,
$a \in \mathcal{E}_a$,
$og_0, og_1 \in \mathcal{E}_{og}$,
$og_0 \neq og_1$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

Transitivity Rules. Like their language $\mathcal{L}^*$ counterparts, these rules ensure that for any three distinct groups g_0, g_1 and g_2, if g_0 is a subset of g_1 and g_1 is a subset of g_2, then g_0 is also a subset of g_2.

1. Subject Group Transitivity Rules

$\forall\ (sg_0, sg_1, sg_2, \iota, \sigma)$,

$\hat{subst}(sg_0, sg_2, \iota, \sigma) \leftarrow$
$\hat{subst}(sg_0, sg_1, \iota, \sigma), \hat{subst}(sg_1, sg_2, \iota, \sigma)$

where
$sg_0, sg_1, sg_2 \in \mathcal{E}_{sg}$,
$sg_0 \neq sg_1 \neq sg_2$,
$\iota \in \mathcal{I}$,
$S_0 \leq \sigma \leq S_{|\psi|}$

2. Access Right Group Transitivity Rules

$\forall\ (ag_0, ag_1, ag_2, \iota, \sigma)$,

$\hat{subst}(ag_0, ag_2, \iota, \sigma) \leftarrow$
$\quad \hat{subst}(ag_0, ag_1, \iota, \sigma), \hat{subst}(ag_1, ag_2, \iota, \sigma)$

where
$\quad ag_0, ag_1, ag_2 \in \mathcal{E}_{ag}$,
$\quad ag_0 \neq ag_1 \neq ag_2$,
$\quad \iota \in \mathcal{I}$,
$\quad S_0 \leq \sigma \leq S_{|\psi|}$

3. Object Group Transitivity Rules

$\forall\ (og_0, og_1, og_2, \iota, \sigma)$,

$\hat{subst}(og_0, og_2, \iota, \sigma) \leftarrow$
$\quad \hat{subst}(og_0, og_1, \iota, \sigma), \hat{subst}(og_1, og_2, \iota, \sigma)$

where
$\quad og_0, og_1, og_2 \in \mathcal{E}_{og}$,
$\quad og_0 \neq og_1 \neq og_2$,
$\quad \iota \in \mathcal{I}$,
$\quad S_0 \leq \sigma \leq S_{|\psi|}$

Inertial Rules. The same rules of inertia expressed in language $\mathcal{L}^*$ applies to language $\mathcal{L}^{\mathcal{T}^*}$: every fact $\hat{\rho}$ that holds in state σ must also hold in state $Res(u, \sigma)$ after policy update u is applied, provided that update u does not cause the fact $\neg\hat{\rho}$ to hold.

$\forall\ (\hat{\alpha}, u)$,

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

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

where
$\quad \hat{\alpha} \in \mathcal{A}^{\sigma}$,
$\quad u \in \psi$,
$\quad \hat{\alpha}'$ = $CopyAtom(\hat{\alpha}, Res(u, \sigma))$

Reflexivity Rules. These rules ensure that the reflexive property of sets are preserved: every set is a subset of itself.

$\forall\ (g, \iota, \sigma)$,

$\hat{subst}(g, g, \iota, \sigma) \leftarrow$

where

$g \in (\mathcal{E}_{sg} \cup \mathcal{E}_{ag} \cup \mathcal{E}_{og})$,

$\iota \in \mathcal{I}$,

$S_0 \leq \sigma \leq S_{|\psi|}$

Temporal Rules. The temporal rules are based on the fact that if a fact $\hat{\rho}$ holds at interval ι, then the same fact $\hat{\rho}$ must also hold at all intervals ι' where the relation between intervals ι and ι' is *Equals*, *During*, *Starts* or *Finishes*.

1. Holds Temporal Rules

   $\forall(s, a, o, \iota_0, \iota_1, \sigma)$,

   $\hat{holds}(s, a, o, \iota_1, \sigma) \leftarrow \hat{holds}(s, a, o, \iota_0, \sigma)$

   $\neg\ \hat{holds}(s, a, o, \iota_1, \sigma) \leftarrow \neg\ \hat{holds}(s, a, o, \iota_0, \sigma)$

   where

   $s \in \mathcal{E}_s$,

   $a \in \mathcal{E}_a$,

   $0 \in \mathcal{E}_o$,

   $\iota_0, \iota_1 \in \mathcal{I}$,

   $NET.Get(\iota_0, \iota_1) \subseteq \{$*During*, *Starts*, *Finishes*, *Equals*$\}$,

   $S_0 \leq \sigma \leq S_{|V|}$

2. Membership Temporal Rules

   $\forall(ss, sg, \iota_0, \iota_1, \sigma)$,

   $\hat{memb}(ss, sg, \iota_1, \sigma) \leftarrow \hat{memb}(ss, sg, \iota_0, \sigma)$

   $\neg\ \hat{memb}(ss, sg, \iota_1, \sigma) \leftarrow \neg\ \hat{memb}(ss, sg, \iota_0, \sigma)$

where

$ss \in \mathcal{E}_{ss}$,

$sg \in \mathcal{E}_{sg}$,

$\iota_0, \iota_1 \in \mathcal{I}$,

$NET.Get(\iota_0, \iota_1) \subseteq \{$*During*, *Starts*, *Finishes*, *Equals*$\}$,

$S_0 \leq \sigma \leq S_{|V|}$

$\forall(as, ag, \iota_0, \iota_1, \sigma)$,

$\hat{memb}(as, ag, \iota_1, \sigma) \leftarrow \hat{memb}(as, ag, \iota_0, \sigma)$

$\neg\, \hat{memb}(as, ag, \iota_1, \sigma) \leftarrow \neg\, \hat{memb}(as, ag, \iota_0, \sigma)$

where

$as \in \mathcal{E}_{as}$,

$ag \in \mathcal{E}_{ag}$,

$\iota_0, \iota_1 \in \mathcal{I}$,

$NET.Get(\iota_0, \iota_1) \subseteq \{$*During*, *Starts*, *Finishes*, *Equals*$\}$,

$S_0 \leq \sigma \leq S_{|V|}$

$\forall(os, og, \iota_0, \iota_1, \sigma)$,

$\hat{memb}(os, og, \iota_1, \sigma) \leftarrow \hat{memb}(os, og, \iota_0, \sigma)$

$\neg\, \hat{memb}(os, og, \iota_1, \sigma) \leftarrow \neg\, \hat{memb}(os, og, \iota_0, \sigma)$

where

$os \in \mathcal{E}_{os}$,

$og \in \mathcal{E}_{og}$,

$\iota_0, \iota_1 \in \mathcal{I}$,

$NET.Get(\iota_0, \iota_1) \subseteq \{$*During*, *Starts*, *Finishes*, *Equals*$\}$,

$S_0 \leq \sigma \leq S_{|V|}$

3. Subset Temporal Rules

$\forall(sg_0, sg_1, \iota_0, \iota_1, \sigma)$,

$\hat{subst}(sg_0, sg_1, \iota_1, \sigma) \leftarrow \hat{subst}(sg_0, sg_1, \iota_0, \sigma)$

$\neg\ \hat{subst}(sg_0, sg_1, \iota_1, \sigma) \leftarrow \neg\ \hat{subst}(sg_0, sg_1, \iota_0, \sigma)$

where

$sg_0, sg_1 \in \mathcal{E}_{sg}$,
$\iota_0, \iota_1 \in \mathcal{I}$,
$NET.Get(\iota_0, \iota_1) \subseteq \{\textit{During}, \textit{Starts}, \textit{Finishes}, \textit{Equals}\}$,
$S_0 \leq \sigma \leq S_{|V|}$

$\forall(ag_0, ag_1, \iota_0, \iota_1, \sigma)$,

$\hat{subst}(ag_0, ag_1, \iota_1, \sigma) \leftarrow \hat{subst}(ag_0, ag_1, \iota_0, \sigma)$

$\neg\ \hat{subst}(ag_0, ag_1, \iota_1, \sigma) \leftarrow \neg\ \hat{subst}(ag_0, ag_1, \iota_0, \sigma)$

where

$ag_0, ag_1 \in \mathcal{E}_{ag}$,
$\iota_0, \iota_1 \in \mathcal{I}$,
$NET.Get(\iota_0, \iota_1) \subseteq \{\textit{During}, \textit{Starts}, \textit{Finishes}, \textit{Equals}\}$,
$S_0 \leq \sigma \leq S_{|V|}$

$\forall(og_0, og_1, \iota_0, \iota_1, \sigma)$,

$\hat{subst}(og_0, og_1, \iota_1, \sigma) \leftarrow \hat{subst}(og_0, og_1, \iota_0, \sigma)$

$\neg\ \hat{subst}(og_0, og_1, \iota_1, \sigma) \leftarrow \neg\ \hat{subst}(og_0, og_1, \iota_0, \sigma)$

where

$og_0, og_1 \in \mathcal{E}_{og}$,
$\iota_0, \iota_1 \in \mathcal{I}$,
$NET.Get(\iota_0, \iota_1) \subseteq \{\textit{During}, \textit{Starts}, \textit{Finishes}, \textit{Equals}\}$,
$S_0 \leq \sigma \leq S_{|V|}$

Example 4.3 *The following rules show the language $\mathcal{L}^{\mathcal{T}^*}$ translation of the language $\mathcal{L}^{\mathcal{T}}$ code listing shown in Example 4.2.*

1. *Initial Fact Rules*

$\hat{memb}$(alice, grp_2, work_hours, S_0) $\leftarrow$
$\hat{subst}$(grp_2, grp_1, morning_hours, S_0) $\leftarrow$
$\hat{holds}$(grp_1, read, file, work_hours, S_0) $\leftarrow$

2. *Authorisation Constraint Rules*

$\hat{holds}$(grp_1, write, file, work_hours, S_0) $\leftarrow$
$\hat{holds}$(grp_1, read, file, work_hours, S_0),
not $\neg$ $\hat{holds}$(grp_3, write, file, work_hours, S_0)

$\hat{holds}$(grp_1, write, file, work_hours, S_1) $\leftarrow$
$\hat{holds}$(grp_1, read, file, work_hours, S_1),
not $\neg$ $\hat{holds}$(grp_3, write, file, work_hours, S_1)

$\vdots$

$\hat{holds}$(grp_1, write, file, afternoon_hours, S_0) $\leftarrow$
$\hat{holds}$(grp_1, read, file, afternoon_hours, S_0),
not $\neg$ $\hat{holds}$(grp_3, write, file, afternoon_hours, S_0)

$\hat{holds}$(grp_1, write, file, afternoon_hours, S_1) $\leftarrow$
$\hat{holds}$(grp_1, read, file, afternoon_hours, S_1),
not $\neg$ $\hat{holds}$(grp_3, write, file, afternoon_hours, S_1)

3. *Policy Update Rules*

$\hat{holds}$(grp_1, read, file, work_hours, S_1) $\leftarrow$
$\hat{holds}$(grp_1, read, file, morning_hours, S_1) $\leftarrow$
$\hat{holds}$(grp_1, read, file, afternoon_hours, S_1) $\leftarrow$

4. *Inheritance Rules*

$\hat{holds}$(grp_1, read, file, work_hours, S_0) $\leftarrow$
$\hat{holds}$(grp_2, read, file, work_hours, S_0),
$\hat{subst}$(grp_1, grp_2, work_hours, S_0),
not $\neg$ $\hat{holds}$(grp_1, read, file, work_hours, S_0)

$\neg\, \hat{holds}(grp_1, read, file, work\_hours, S_0) \leftarrow$
$\neg\, \hat{holds}(grp_2, read, file, work\_hours, S_0),$
$\hat{subst}(grp_1, grp_2, work\_hours, S_0)$

$\vdots$

$\hat{holds}(grp_3, write, file, afternoon\_hours, S_1) \leftarrow$
$\hat{holds}(grp_2, write, file, afternoon\_hours, S_1),$
$\hat{subst}(grp_3, grp_2, afternoon\_hours, S_1),$
$not\ \neg\, \hat{holds}(grp_3, write, file, afternoon\_hours, S_1)$

$\neg\, \hat{holds}(grp_3, write, file, afternoon\_hours, S_1) \leftarrow$
$\neg\, \hat{holds}(grp_2, write, file, afternoon\_hours, S_1),$
$\hat{subst}(grp_3, grp_2, afternoon\_hours, S_1)$

$\hat{holds}(alice, read, file, work\_hours, S_0) \leftarrow$
$\hat{holds}(grp_1, read, file, work\_hours, S_0),$
$\hat{memb}(alice, grp_1, work\_hours, S_0),$
$not\ \neg\, \hat{holds}(alice, read, file, work\_hours, S_0)$

$\neg\, \hat{holds}(alice, read, file, work\_hours, S_0) \leftarrow$
$\neg\, \hat{holds}(grp_1, read, file, work\_hours, S_0),$
$\hat{memb}(alice, grp_1, work\_hours, S_0)$

$\vdots$

$\hat{holds}(alice, write, file, afternoon\_hours, S_1) \leftarrow$
$\hat{holds}(grp_3, write, file, afternoon\_hours, S_1),$
$\hat{memb}(alice, grp_3, afternoon\_hours, S_1),$
$not\ \neg\, \hat{holds}(alice, write, file, afternoon\_hours, S_1)$

$\neg\, \hat{holds}(alice, write, file, afternoon\_hours, S_1) \leftarrow$
$\neg\, \hat{holds}(grp_3, write, file, afternoon\_hours, S_1),$
$\hat{memb}(alice, grp_3, afternoon\_hours, S_1)$

5. Transitivity Rules

$\hat{subst}(grp_1, grp_3, work\_hours, S_0) \leftarrow$
$\hat{subst}(grp_1, grp_2, work\_hours, S_0)$,
$\hat{subst}(grp_2, grp_3, work\_hours, S_0)$

$\vdots$

$\hat{subst}(grp_3, grp_1, afternoon\_hours, S_1) \leftarrow$
$\hat{subst}(grp_3, grp_2, afternoon\_hours, S_1)$,
$\hat{subst}(grp_2, grp_1, afternoon\_hours, S_1)$

6. Inertial Rules

$\hat{holds}(alice, read, file, work\_hours, S_1) \leftarrow$
$\hat{holds}(alice, read, file, work\_hours, S_0)$,
$not \neg \hat{holds}(alice, read, file, work\_hours, S_1)$

$\vdots$

$\hat{holds}(grp_3, write, file, afternoon\_hours, S_1) \leftarrow$
$\hat{holds}(grp_3, write, file, afternoon\_hours, S_0)$,
$not \neg \hat{holds}(grp_3, write, file, afternoon\_hours, S_1)$

$\hat{memb}(alice, grp_1, work\_hours, S_1) \leftarrow$
$\hat{memb}(alice, grp_1, work\_hours, S_0)$,
$not \neg \hat{memb}(alice, grp_1, work\_hours, S_1)$

$\vdots$

$\hat{memb}(alice, grp_3, afternoon\_hours, S_1) \leftarrow$
$\hat{memb}(alice, grp_3, afternoon\_hours, S_0)$,
$not \neg \hat{memb}(alice, grp_3, afternoon\_hours, S_1)$

$\hat{subst}(grp_1, grp_1, work\_hours, S_1) \leftarrow$
$\hat{subst}(grp_1, grp_1, work\_hours, S_0)$,
$not \neg \hat{subst}(grp_1, grp_1, work\_hours, S_1)$

$\vdots$

$\hat{subst}(grp_3, grp_3, afternoon\_hours, S_1) \leftarrow$
$\hat{subst}(grp_3, grp_3, afternoon\_hours, S_0)$,
$not \neg \hat{subst}(grp_3, grp_3, afternoon\_hours, S_1)$

7. *Reflexivity Rules*

$\hat{subst}(grp_1, grp_1, work\_hours, S_0) \leftarrow$

$\vdots$

$\hat{subst}(grp_3, grp_3, afternoon\_hours, S_1) \leftarrow$

8. *Temporal Rules*

$\hat{holds}(alice, read, file, morning\_hours, S_0) \leftarrow$
$\hat{holds}(alice, read, file, work\_hours, S_0)$,

$\neg \hat{holds}(alice, read, file, morning\_hours, S_0) \leftarrow$
$\neg \hat{holds}(alice, read, file, work\_hours, S_0)$,

$\vdots$

$\hat{holds}(grp_3, write, file, afternoon\_hours, S_1) \leftarrow$
$\hat{holds}(grp_3, write, file, work\_hours, S_1)$,

$\neg \hat{holds}(grp_3, write, file, afternoon\_hours, S_1) \leftarrow$
$\neg \hat{holds}(grp_3, write, file, work\_hours, S_1)$,

$\hat{memb}(alice, grp_1, morning\_hours, S_0) \leftarrow$
$\hat{memb}(alice, grp_1, work\_hours, S_0)$,

$\neg \hat{memb}(alice, grp_1, morning\_hours, S_0) \leftarrow$
$\neg \hat{memb}(alice, grp_1, work\_hours, S_0)$,

$\vdots$

$\hat{memb}(alice, grp_3, afternoon\_hours, S_1) \leftarrow$
$\hat{memb}(alice, grp_3, work\_hours, S_1)$,

$$\neg\, \hat{memb}(alice,\, grp_3,\, afternoon\_hours,\, S_1) \leftarrow$$
$$\quad \neg\, \hat{memb}(alice,\, grp_3,\, work\_hours,\, S_1),$$

$$\hat{subst}(grp_1,\, grp_2,\, morning\_hours,\, S_0) \leftarrow$$
$$\quad \hat{subst}(grp_1,\, grp_2,\, work\_hours,\, S_0),$$

$$\neg\, \hat{subst}(grp_1,\, grp_2,\, morning\_hours,\, S_0) \leftarrow$$
$$\quad \neg\, \hat{subst}(grp_1,\, grp_2,\, work\_hours,\, S_0),$$

$$\vdots$$

$$\hat{subst}(grp_3,\, grp_2,\, afternoon\_hours,\, S_1) \leftarrow$$
$$\quad \hat{subst}(grp_3,\, grp_2,\, work\_hours,\, S_1),$$

$$\neg\, \hat{subst}(grp_3,\, grp_2,\, afternoon_hours,\, S_1) \leftarrow$$
$$\quad \neg\, \hat{subst}(grp_3,\, grp_2,\, work\_hours,\, S_1),$$

4.5 Discussions

The previous section has shown how a language $\mathcal{L}^{\mathcal{T}}$ program can be translated into a extended logic program language $\mathcal{L}^{\mathcal{T}^*}$. Since language $\mathcal{L}^{\mathcal{T}^*}$ is semantically similar to language $\mathcal{L}^*$, the same methods shown in Section 2.2.3 is used to translate language $\mathcal{L}^{\mathcal{T}^*}$ into a normal logic program for query evaluation. However, one outstanding issue remains for language $\mathcal{L}^{\mathcal{T}}$: domain consistency.

A language $\mathcal{L}^{\mathcal{T}}$ domain description $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ without any variable occurrences (and therefore without any *where* clauses) may be represented as the following statements:

```
initially ρ0_0, ..., ρ0_n0, !ρ1_0, ..., !ρ1_n1;

relation r0_0, ..., r0_m;

always ρ2_0, ..., ρ2_n2, !ρ3_0, ..., !ρ3_n3
  implied by ρ4_0, ..., ρ4_n4, !ρ5_0, ..., !ρ5_n5
  with absence ρ6_0, ..., ρ6_n6, !ρ7_0, ..., !ρ7_n7;

update u()
  causes ρ8_0, ..., ρ8_n8, !ρ9_0, ..., !ρ9_n9
  if ρ10_0, ..., ρ10_n10, !ρ11_0, ..., !ρ11_n11;
```

Note that if we disregard the *relation* statement and the fact that language $\mathcal{L}^{\mathcal{T}}$ atoms also specify time intervals, language $\mathcal{L}^{\mathcal{T}}$ statements consisting only of ground expressions as above are identical to the language $\mathcal{L}$ statements shown in Section 2.3. As a consequence, Theorem 2.2 can be used to define $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ consistency:

Definition 4.4 *A domain description $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ is* **consistent** *if:*

1. *$\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ is* **normal***, using the same language $\mathcal{L}$ definition in Definition 2.7; and*
2. *The temporal interval relations defined in $\mathcal{D}_{\mathcal{L}^{\mathcal{T}}}$ produces a* **consistent interval network**.

Assuming that we already have a normal domain description, we focus on the second condition. Below, we define network consistency:

Definition 4.5 *Given a temporal interval network N, with nodes n_0, ... n_n and arcs a_0, ..., a_m. An* **instance** *of interval network N is a subnetwork with nodes n_0, ..., n_n, with each arc a'_i representing a single relation from the corresponding arc in N ($a'_i \in a_i$, $0 \leq i \leq m$). A* **consistent instance** *of interval network N is an instance of N such that for any three nodes, the three relations represented by these nodes satisfy the transitivity rules shown in Table 4.1. A* **consistent interval network** *is a network with at least one consistent instance.*

Any given language $\mathcal{L}^{\mathcal{T}}$ domain description can fail to satisfy Condition 2 in Definition 4.4 above in different ways. In fact, Valdez-Perez [60] generalised these conditions where an interval network is inconsistent[1]. One such condition arises in language $\mathcal{L}^{\mathcal{T}}$ when two or more conflicting interval constraint statements exist. This conflict occurs when the consequence of an interval constraint is implicitly or explicitly excluded by another interval constraint. For example, the following two interval constraint statements are in conflict with each other because the second defines a relation that has been explicitly excluded by the first.

```
relation during(i0, i1), before(i0, i1);
relation equals(i0, i1);
```

Note that this type of domain inconsistency is detected by the $NET.AddRel()$ algorithm. Unfortunately, the second type of inconsistency is not. To illustrate this type of inconsistency, we use Allen's example in [2], shown in Figure 4.8.

The network shown in Figure 4.8 is in fact inconsistent, as none of its 32 instances are consistent. To see this, we note that although the $NET.AddRel()$ algorithm accepts the

[1] Valdez-Perez uses the term *unsatisfiable*.

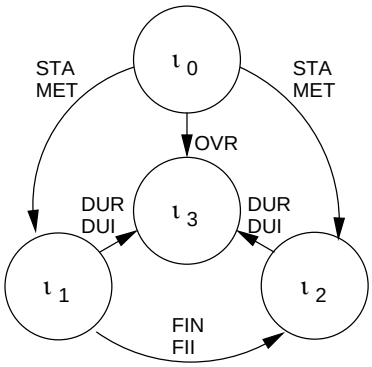

Figure 4.8: Inconsistent Network

relation shown between the arcs i_1 and i_2, we cannot assign the relation *finishes* or *finished by* alone.

In fact, as Allen points out in [2], the $NET.AddRel()$ algorithm can only guarantee consistency within any three nodes of the network (*3-consistency*). Unfortunately, Vilain et al. [63] points out that algorithms that can ensure full network consistency, i.e. *k-consistency* [26], are in fact, intractable. Nevertheless, total consistency can be easily checked by employing a simple backtracking algorithm. Furthermore, several methods for full network consistency checking, like those in [36, 62] already exist. In this work, we make the assumption that 3-consistency is sufficient for our authorisation system as we have observed that this covers all scenarios that we have examined.

In this chapter, we have introduced an algebra for expressing time interval relations and an authorisation language that employs this algebra to express temporal constraints. In the next chapter, we shall describe in detail the implementation of an authorisation system whose policies are expressed in this language.

Chapter 5

Implementation Issues

This chapter focuses on two issues: (1) the implementation of *PolicyUpdater* version 2, which includes support for temporal constraints; and (2) an in-depth discussion of implementation issues not previously covered in the other chapters.

The source code and other technical information of all the versions of the PolicyUpdater core system, *Vlad*, is available in the project website:

```
http://www.scm.uws.edu.au/~jcrescin/projects/policyupdater/index.html
```

The PolicyUpdater software package makes use of a temporal reasoner engine to evaluate temporal interval relations under the rules of the interval algebra discussed in Chapter 4. The source code and other implementation details of a full library implementation of the temporal reasoner engine, *Tribe*, can be found in the following website:

```
http://www.scm.uws.edu.au/~jcrescin/projects/tribe/index.html
```

5.1 System Structure

Figure 5.1 shows the internal processes of the system. The overall process of the system is similar to that of the previous version:

An access control policy, written in language $\mathcal{L}^{\mathcal{T}}$ syntax, is translated into a normal logic program, which in turn is fed to *SModels* to generate answer sets. From these answer sets, the system can evaluate authorisation queries.

What makes PolicyUpdater 2 different from its predecessor is that it uses a temporal reasoner to generate a normal logic program. The system gathers all temporal intervals and relations from the policy and makes use of the temporal reasoner to derive new interval relations not explicitly stated in the policy.

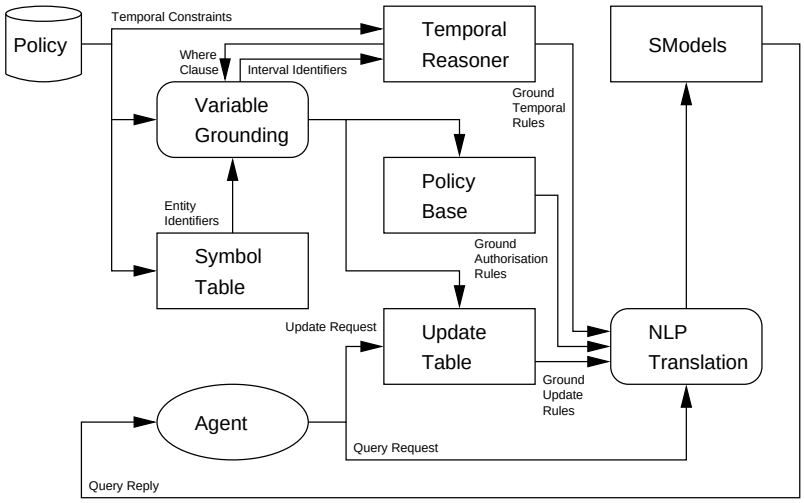

Figure 5.1: System Flowchart

Another difference is the variable grounding process. With PolicyUpdater 1, variables are resolved just before they are stored in the policy base. This is possible because in PolicyUpdater 1, the only variables that occur in the policy are resolved to all entities defined that are of the same type. However, in PolicyUpdater 2, the intervals used to ground interval variables are subject to some restrictions defined by *where* clauses in policy statements. As a result, the variable grounding process of PolicyUpdater 2 needs some interaction with the temporal reasoner engine.

The internal mechanisms of both the temporal reasoner engine and the policy base engine shall be discussed in detail in the following sections. However, before we do so, we first formalise a list structure first introduced in Section 3.1.2 as the implementation of both engines makes use of this structure.

Definition 5.1 *A list object $LIST$ is defined as a linked-list implementation with the following operators:*

1. $LIST.Init()$ *initialises or re-initialises* $LIST$*. Any items in the list is removed after this operator is called.*
2. $LIST.Length()$ *returns the number of items stored in* $LIST$*. The shorthand notation* $|LIST|$ *is equivalent to* $LIST.Length()$*.*
3. $LIST.Find(item)$ *returns* true *if* $item$ *is stored in* $LIST$*,* false *otherwise.*
4. $LIST.Index(item)$ *returns the ordinal index of* $item$ *with respect to* $LIST$*.*

5. *$LIST.Add(item)$ attaches $item$ to the end of $LIST$.*

6. *$LIST.Del(index)$ removes the $index$'th item from $LIST$.*

7. *$LIST.Get(index)$ returns the $index$'th item in $LIST$. The shorthand notation $LIST[index]$ is equivalent to $LIST.Get(index)$.*

8. *$LIST.Append(list)$ attaches the items in $list$ at the end of $LIST$.*

5.2 Temporal Reasoner

The primary purpose of the temporal reasoner engine is to maintain a complete network of temporal intervals and their relations to each other, as discussed in Chapter 4, such that: (1) the effects of adding a new relation are propagated to the entire network; and (2) the relation between any two intervals can be looked up at any given time. This section focuses on the methods used in the implementation of this reasoner engine.

5.2.1 Network Structure

Conceptually, a temporal interval relation network may be represented as a table whose rows and columns both represent all the intervals in the network. As shown in Table 5.1, all the possible relations that exist between intervals ι_i and ι_j, as stored in the network, are contained in the relation set RS_{ij} from cell (i, j).

	ι_0	ι_1	...	ι_n
ι_0	RS_{00}	RS_{01}		RS_{0n}
ι_1	RS_{10}	RS_{11}		RS_{1n}
⋮				⋮
ι_n	RS_{n0}	RS_{n1}	...	RS_{nn}

Table 5.1: Conceptual Representation of an Interval Network

As we shall see, using this representation as a basis for implementation is not a good idea, as it contains redundant information and will incur a high maintenance overhead. First, we note that due to the reflexive property of intervals, all the cells in the diagonal of this representation is defined as follows:

$\forall\, i,$

$RS_{ii} = EQL$

where

$0 \leq i \leq n$

As the reflexive property applies to all intervals, there is no need to store such relations in the network. Another instance of redundancy in this representation is made evident by the following property:

$\forall\, (i, j)$,

$RS_{ij} = RS_{ji}^{-1}$

where

$0 \leq i \leq n$

$0 \leq j \leq n$

RS^{-1} is the inverse set of RS

The property above states that a relation set between two intervals ι_0 and ι_1 can be derived by calculating the inverse of the relation set between ι_1 and ι_0. As a consequence, only one relation set between any two intervals needs to be stored as its inverse can be calculated from this relation set if required. Note that due to this property, half of the cells in Table 5.1 are redundant. The information stored above the diagonal is the inverse of those stored below.

Binary Representation of Relation Sets

Obviously, relation sets can be conceptually represented as a list of relations. However, from an implementation point of view, maintaining a list of relations for every relation set can incur a lot of overhead as the cost of adding and removing relations from such a list can be computationally expensive. To avoid this problem, we employ a more efficient way of representing these sets in our implementation. Each relation set is represented as a bit vector[1].

To illustrate this representation, we first assign a base-2 value for each basic relation as shown in Table 5.2. With these value assignments, any relation set composed of any combination of these basic relations can be represented as an integer which is the sum of the relations that it contains. For example, under this system the relation set $\{BEF, DUR, STA, STI\}$ can be represented as 2086.

Implemented as a bit vector, three important relation set operators can be easily defined by using bitwise operators:

[1] A similar method is used in [62, 63].

Relation	Symbol	Value	Bit Value
Equals	EQL	1	0
Before	BEF	2	1
During	DUR	4	2
Overlaps	OVR	8	3
Meets	MET	16	4
Starts	STA	32	5
Finishes	FIN	64	6
After	BEI	128	7
Contains	DUI	256	8
Overlapped By	OVI	512	9
Met By	MEI	1024	10
Started By	STI	2048	11
Finished By	FII	4096	12

Table 5.2: Temporal Relation Value Assignment

- The *Union* between two relation sets is calculated by performing a *bitwise and* operation between the two sets.

  $$RS_0 \cup RS_1 = RS_0 \mid RS_1$$

- The *Intersection* between two relation sets is simply the opposite of the union operation: a *bitwise or* operation between the two sets.

  $$RS_0 \cap RS_1 = RS_0 \ \& \ RS_1$$

- The *Inverse* of a relation set can be calculated by replacing each relation bit, as defined by Table 5.2, by its inverse relation bit. This is achieved by a series of bit masking and shifting operations:

  $$RS^{-1} = ((RS \ \& \ 8064) >> 6) \mid ((RS \ \& \ 126) << 6) \mid (RS \ \& \ 1)$$

Network Data Structure

The temporal interval network is implemented as a list of all intervals, each with a relation list where information about its relation with other intervals are stored (see Figure 5.2). Table 5.3 shows the structure of each node in the network list, while Table 5.4 shows the structure of each relation list node in each network list node.

For example, the network shown in Figure 5.3 contains only the relations $before(\iota_0, \iota_1)$ and $during(\iota_0, \iota_1)$. This network, NET, is stored in the following manner:

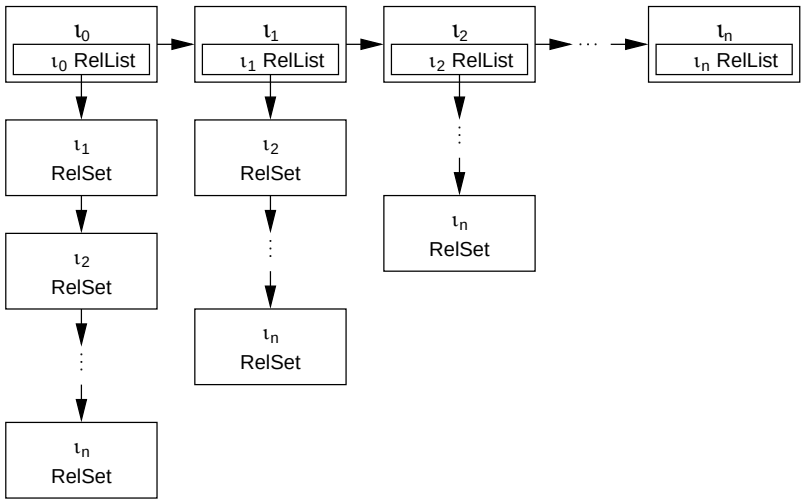

Figure 5.2: Network Structure as a List of Relation Lists

Field	**Type**	**Description**
int	String	Interval ID
ep_0	Integer	Starting End Point
ep_1	Integer	Finishing End Point
$rlist$	RelList	Relation List

Table 5.3: Network Node Data Structure

Field	**Type**	**Description**
int	String	Interval ID
rs	Integer	Relation Set

Table 5.4: Relation List Node Data Structure

$NET[0].int = \iota_0$

$NET[0].rlist[0].int = \iota_1$

$NET[0].rlist[0].rs = 6$

$NET[1].int = \iota_1$

$NET[1].rlist = \emptyset$

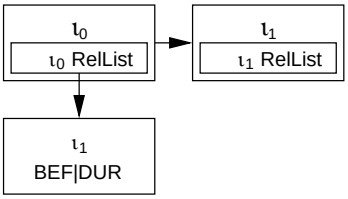

Figure 5.3: Network Structure Containing $before(\iota_0, \iota_1)$ and $during(\iota_0, \iota_1)$

Note that in the previous example, the relations between intervals ι_0 and ι_1 are stored in the relation list of interval ι_0, but the inverse relations, those between intervals ι_1 and ι_0, are not stored in the relation list of ι_1. This is done for efficiency, as inverse relations can be calculated by the inverse operator for any stored relations.

However, this method raises the question of how we decide which interval node is used to store a given relation. In the previous example, we arbitrarily chose to store the relation in interval ι_0's relation list, however, as shown in Figure 5.4, we could just as easily have chosen to store the relation in interval ι_1's relation list:

$NET[0].int = \iota_0$

$NET[0].rlist = \emptyset$

$NET[1].int = \iota_1$

$NET[1].rlist[0].int = \iota_0$

$NET[1].rlist[0].rs = 384$

To resolve this issue, we adopt a simple rule to determine where a given relation is to be stored in the network. Given a pair of intervals ι_0 and ι_1, and a relation set rs, this relation is stored in the node whose interval ID occurs first when sorted alphabetically. For example, if ι_0 is $morning$ and ι_1 is $evening$, then the relation rs would be stored in the node containing ι_1, since $evening$ is alphabetically before $morning$. To enforce this rule, we use the function shown in Algorithm 5.1 before adding a relation to the network.

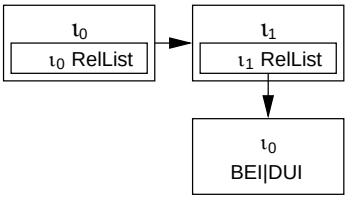

Figure 5.4: Equivalent Representation of $before(\iota_0, \iota_1)$ and $during(\iota_0, \iota_1)$

Algorithm 5.1 *NormaliseRel()*

```
FUNCTION NormaliseRel(ι0, ι1, rs)
  IF ι0 ≤ ι1 THEN
    RETURN (ι0, ι1, rs)
  ELSE
    RETURN (ι1, ι0, rs^-1)
  ENDIF
ENDFUNCTION
```

Another issue to be considered arises from the network's completeness property from Definition 4.1, i.e. every node in the network must be connected to every other node. As mentioned in Section 4.2.3, this property is maintained even between nodes containing intervals with no defined relations by assigning the default arc, which represents a relation set containing all 13 relations. Now, considering that the network is implemented as a list of intervals, each with its own list of relations with other intervals, it is easy to see that storing these default relations will incur a lot of overhead. When a new interval is added to the network, a relation list containing all the other intervals must also be added, and each node in this relation list contains the default relation.

Figure 5.5 shows how such implementation might represent a network with 3 intervals and the relations *starts* or *finishes* between intervals ι_0 and ι_1. Note that although only the relation between intervals ι_0 and ι_1 is given, the default relation is still stored between intervals ι_0 and ι_2 and between intervals ι_1 and ι_2.

This issue is easily resolved by simply not storing default relations. In other words, an interval that is not in the relation list of another interval is assumed to have the default relation with that interval. By adopting this technique, we can be certain that the network implementation will only store the minimum information required, and yet the relations between any two intervals can be derived. Figure 5.6 shows how the network in the above example is stored by using this method.

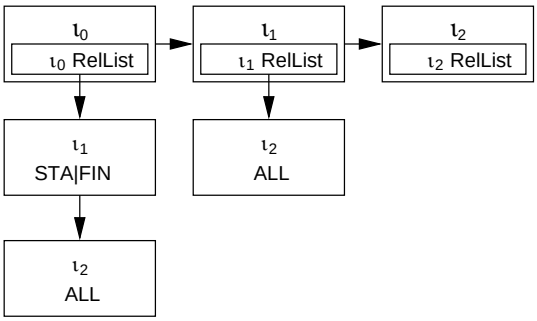

Figure 5.5: Network Structure with Default Relations Stored

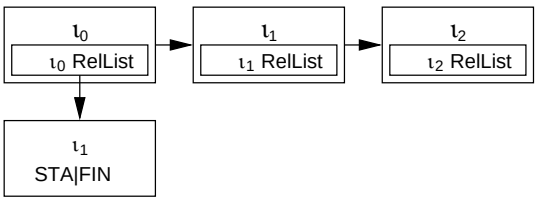

Figure 5.6: Network Structure with Default Relations Omitted

5.2.2 Network Operators

The following defines the full operator set allowed for the interval network. This definition supersedes the basic operators shown in Definition 4.2.

Definition 5.2 *A temporal interval relation network NET implements the following operators:*

1. *The $NET.Init()$ operator initialises or re-initialises the network.*
2. *The $NET.AddInt(\iota)$ operator causes the given interval identifier ι to be added to the network.*
3. *The $NET.AddRel(\iota_0, \iota_1, rs)$ operator adds the relation set rs to the network as the set of possible relations between intervals ι_0 and ι_1, and propagates the effects of this relation to the entire network.*
4. *The $NET.Bind(\iota, ep_0, ep_1)$ operator registers the points ep_0 and ep_1 as the end points of interval ι, then calculates the relation of this interval with other intervals based on their end points.*
5. *The $NET.Get(\iota_0, \iota_1)$ operator returns the relation set between the intervals ι_0 and ι_1.*

We note that the network operators described in Definition 5.2 do not include the operator $NET.Replace()$. This is due to the fact that this is an internal operator used only by the operator $NET.AddRel()$. Nevertheless, the details of how this operator replaces the network relation between the two given intervals by the given relation set is shown in Algorithm 5.2.

Algorithm 5.2 *NET.Replace()*

```
FUNCTION NET.Replace(ι0, ι1, rs)
  (ι0', ι1', rs) = NormaliseRel(ι0, ι1, rs)
  index0 = NET.Index(ι0')
  IF NET[index0].rlist.Find(ι1') THEN
    index1 = NET[index0].rlist.Index(ι1')
    NET[index0].rlist[index1].rs = rs'
  ELSE
    rlnode.int = ι1'
    rlnode.rs = rs'
    NET[index0].rlist.Add(rlnode)
  ENDIF
ENDFUNCTION
```

The $NET.Init()$ operator is a simple routine that initialises the network when called for the first time, and otherwise deletes all the nodes in the network, including all relation list nodes within these network nodes.

The $NET.AddInt(\iota)$ operator is shown in Algorithm 5.3. It works by firstly checking whether a network node with the interval ι is already defined. If not, a new network node is allocated and appended to the end of the network list. Since the NET structure is implemented as a list, the $LIST.Add()$ can be used to add nodes.

Algorithm 5.3 *NET.AddInt()*

```
FUNCTION NET.AddInt(ι)
  IF ι is not in NET THEN
    nnode.int = ι
    nnode.rlist = ∅
    NET.Add(nnode)
  ENDIF
ENDFUNCTION
```

The $NET.AddRel()$ operator shown in Algorithm 4.2 illustrates the relation propagation algorithm based solely on Allen's $ToAdd()$ function in [2]. Algorithm 5.4 shows a simplified algorithm without the using a queue structure. This is made possible by the fact

that the order in which relations are propagated makes no difference in the resulting network [43, 61]. While the algorithm that uses a queue structure performs breadth-first propagation, the recursive algorithm below performs depth-first propagation.

Algorithm 5.4 *NET.AddRel()*

```
FUNCTION NET.AddRel(ι0, ι1, rs)
  IF rs ⊂ NET.Get(ι0, ι1) THEN
    NET.Replace(ι0, ι1, rs)
    FOR each interval ι' ∈ NET DO
      IF ι' != ι0 AND ι' != ι1 THEN
        rs' = NET.Get(ι', ι0)
        IF NOT Skip(rs', rs) THEN
          NET.AddRel(ι', ι1, NET.Get(ι', ι1) ∩ Trans2(rs', rs))
        ENDIF
      ENDIF
    ENDDO
    FOR each interval ι' ∈ NET DO
      IF ι' != ι0 AND ι' != ι1 THEN
        rs' = NET.Get(ι1, ι')
        IF NOT Skip(rs, rs') THEN
          NET.AddRel(ι0, ι', NET.Get(ι0, ι') ∩ Trans2(rs, rs'))
        ENDIF
      ENDIF
    ENDDO
  ENDIF
ENDFUNCTION
```

Algorithm 5.4 also takes advantage of Van Beek and Manchak's observation that certain relation set combinations will not yield new information [62]. These combinations are summarised in the $Skip()$ function shown in Algorithm 5.5.

Algorithm 5.5 *Skip()*

```
FUNCTION Skip(rs0, rs1)
  IF (rs0 == {ALL}) OR
     (rs1 == {ALL}) OR
     (BEF ∈ rs0 AND BEI ∈ rs1) OR
     (BEI ∈ rs0 AND BEF ∈ rs1) OR
     (DUR ∈ rs0 AND DUI ∈ rs1) THEN
    RETURN TRUE
  ENDIF
  RETURN FALSE
```

```
ENDFUNCTION
```

Finally, as the details of the $NET.Bind()$ operator is already shown in Algorithm 4.4, we show the $NET.Get()$ operator in Algorithm 5.6.

Algorithm 5.6 *NET.Get()*

```
FUNCTION NET.Get(ι0, ι1)
  IF ι0 ≤ ι1 THEN
    index = NET.Index(ι0)
    IF NET[index].rlist.Find(ι1) THEN
      RETURN NET[index].rlist[NET[index].rlist.Index(ι1)].rs
    ELSE
      RETURN ALL
    ENDIF
  ELSE
    index = NET.Index(ι1)
    IF NET[index].rlist.Find(ι0) THEN
      RETURN NET[index].rlist[NET[index].rlist.Index(ι0)].rs^-1
    ELSE
      RETURN ALL
    ENDIF
  ENDIF
ENDFUNCTION
```

5.3 Policy Base Engine

The focus of this section is the internal mechanisms of the policy base and other implementation issues. In essence, the policy base is the core authorisation engine of the PolicyUpdater system. While Chapter 3 gave a general overview of the system data structures and processes, here we shall attempt to provide a more detailed formalisation of these internal mechanisms, and to show how the policy base makes use of the temporal reasoner engine.

5.3.1 Data Structures

The policy base is composed of different data structures on which its operators are performed. Table 5.5 summarises these structures. These structures are used by the policy base to store the different components of the policy prior to the normal logic program translation. Note that because of the similarities of PolicyUpdater version 1 and version 2, some of the more fundamental structures have already been defined in Section 3.1.2.

Field	Type	Description
$symtab$	Symbol Table	List of Entity and Interval Identifiers
$inittab$	Expression	List of Initial Facts
$consttab$	Constraints Table	List of Constraints
$updatetab$	Update Declaration Table	List of Policy Update Declarations
$seqtab$	Update Sequence Table	List of Policy Update Declarations
$netwk$	Relation Network	Interval Relation Network

Table 5.5: Policy Base Structure

The symbol table $PB.symtab$, whose structure is shown in Table 5.6, is used by the policy base to store identifiers. The main difference of this symbol table to that of version 1 (shown in Table 3.1) is that it also stores interval identifiers. However, the general structure remains the same: each field is a list containing all defined entities and intervals of a specific type.

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
s	String List	$ss + sg$
a	String List	$as + ag$
o	String List	$os + og$
int	String List	Interval

Table 5.6: Extended Symbol Table

The initial facts table $PB.inittab$ is a list of all facts declared in the language $\mathcal{L}^{\mathcal{T}}$ *initially* statement. Recall from Section 3.1.2 that the fact structure is composed of an atom, a type specifier $type \in \{h, m, s\}$ and a truth indicator $truth \in \{true, false\}$. Table 5.7 shows the data structure used to store atoms. Note that the atom structure is extended to include interval identifiers for each atom type.

The constraints table $PB.consttab$ is used to store authorisation constraints. The table is implemented as a list where each item is composed of the following fields: three authorisation expressions (the consequent, and the positive and negative premises) and an interval relation expression. Table 5.8 shows the structure of each node of this list.

The interval relation list used by the constraints table is a list whose nodes are composed

Atom	**Field**	**Type**	**Description**
holds	sub	String	Subject Entity
	acc	String	Access Right Entity
	obj	String	Object Entity
	int	String	Interval
member	elt	String	Single Entity
	grp	String	Group Entity
	int	String	Interval
	$type$	$\{sub\|acc\|obj\}$	Type Specifier
subset	grp_0	String	Subgroup Entity
	grp_1	String	Supergroup Entity
	int	String	Interval
	$type$	$\{sub\|acc\|obj\}$	Type Specifier

Table 5.7: Extended Atom Data Structure

Field	**Type**	**Description**
exp	Expression Type	Consequent
pcn	Expression Type	Positive Premise
ncn	Expression Type	Negative Premise
$rexp$	Interval Relation List	*where* Clause

Table 5.8: Constraints Table Node

of two intervals and a relation set. The details of this structure is shown in Table 5.9.

Field	Type	Description
int_0	String	Interval
int_1	String	Interval
rs	Integer	Relation Set

Table 5.9: Interval Relation List Node

The structure of each field in the policy update declarations table $PB.updatetab$ is shown in Table 5.10. This table is used by the policy base to store all policy update declarations. The policy update sequence table $PB.seqtab$ is a list whose node structure is the same as those already shown in Table 3.6.

Field	Type	Description
$name$	String	Update Identifier
$vlist$	Ordered String List	Variables
pre	Expression Type	Precondition
pst	Expression Type	Postcondition
$rexp$	Interval Relation List	*where* Clause

Table 5.10: Policy Update Declarations Table Node

Lastly, the relation network field $PB.netwk$ is an instance of the interval relation network object described in Definition 5.2.

5.3.2 Encoding Atoms

Recall that the $Encode()$ function first introduced in Section 3.2.3 takes as input an atom α, a state σ and a boolean term τ to indicate whether the fact is classically negated or not, and returns a positive integer i unique to these parameters:

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

For example, given the fact $holds(alice, exec, file, today)$ which holds at state S_0 may be assigned the integer 0 by the $Encode$ function. Similarly, the negation of the fact may be assigned the integer 1, and so on:

$$0 = Encode(holds(alice, exec, file, today), S_0, true)$$

$$1 = Encode(holds(alice, exec, file, today), S_0, false)$$

$$2 = Encode(holds(alice, exec, file, today), S_1, true)$$

$3 = Encode(holds(alice, exec, file, today), S_1, false)$

Obviously, a consistent mapping between these integers and these facts are needed, and a systematic representation of facts is required to achieve and maintain this consistent mapping. Table 5.11 shows a conceptual systematic arrangement of facts using the symbol table θ shown in Table 5.6.

		Entity 1	**Entity 2**	**Entity 3**	**Interval**
Holds		$\theta.s[0]$ ⋮ $\theta.s[\|\theta.s\|\text{-}1]$	$\theta.a[0]$ ⋮ $\theta.a[\|\theta.a\|\text{-}1]$	$\theta.o[0]$ ⋮ $\theta.o[\|\theta.o\|\text{-}1]$	$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Sub	**Member**	$\theta.ss[0]$ ⋮ $\theta.ss[\|\theta.ss\|\text{-}1]$	$\theta.sg[0]$ ⋮ $\theta.sg[\|\theta.sg\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Acc	**Member**	$\theta.as[0]$ ⋮ $\theta.as[\|\theta.as\|\text{-}1]$	$\theta.ag[0]$ ⋮ $\theta.ag[\|\theta.ag\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Obj	**Member**	$\theta.os[0]$ ⋮ $\theta.os[\|\theta.os\|\text{-}1]$	$\theta.og[0]$ ⋮ $\theta.og[\|\theta.og\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Sub	**Subset**	$\theta.sg[0]$ ⋮ $\theta.sg[\|\theta.sg\|\text{-}1]$	$\theta.sg[0]$ ⋮ $\theta.sg[\|\theta.sg\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Acc	**Subset**	$\theta.ag[0]$ ⋮ $\theta.ag[\|\theta.ag\|\text{-}1]$	$\theta.ag[0]$ ⋮ $\theta.ag[\|\theta.ag\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$
Obj	**Subset**	$\theta.og[0]$ ⋮ $\theta.og[\|\theta.og\|\text{-}1]$	$\theta.og[0]$ ⋮ $\theta.og[\|\theta.og\|\text{-}1]$		$\theta.int[0]$ ⋮ $\theta.int[\|\theta.int\|\text{-}1]$

Table 5.11: Conceptual Arrangement of Facts

Table 5.11 shows all the possible combinations of all entity and interval identifiers defined in the symbol table to form all possible facts in one state. Note that the order in which the facts are enumerated may be used as an index for each fact. For example, using the last single access right $\theta.as[|\theta.as|\text{-}1]$, the last group access right $\theta.ag[|\theta.ag|\text{-}1]$ and the last interval $\theta.int[|\theta.int|\text{-}1]$ from the symbol table, we form the following fact:

$memb(\theta.as[|\theta.as|\text{-}1], \theta.ag[|\theta.ag|\text{-}1], \theta.int[|\theta.int|\text{-}1])$

The index of the fact above can be derived by considering the enumerated order of this fact from Table 5.11, and the fixed order of the symbol table θ:

$$(|\theta.s| \times |\theta.a| \times |\theta.o| \times |\theta.int|) +$$
$$(|\theta.ss| \times |\theta.sg| \times |\theta.int|) +$$
$$(|\theta.as| \times |\theta.ag| \times |\theta.int|) - 1$$

Note that in Table 5.11, the indices of positive member facts are offset by the total number of positive holds facts ($|\theta.s| \times |\theta.a| \times |\theta.o| \times |\theta.int|$) and the indices of positive access right member facts are offset by the total number of positive holds facts plus the total number of positive subject member facts ($|\theta.s| \times |\theta.a| \times |\theta.o| \times |\theta.int| + |\theta.ss| \times |\theta.sg| \times |\theta.int|$). For notational simplicity, we define the following totals of atoms:

$$Tot_A = Tot_H + Tot_M + Tot_S$$

$$Tot_H = |\theta.s| \times |\theta.a| \times |\theta.o| \times |\theta.int|$$

$$Tot_M =$$
$$(|\theta.ss| \times |\theta.sg| \times |\theta.int|) +$$
$$(|\theta.as| \times |\theta.ag| \times |\theta.int|) +$$
$$(|\theta.os| \times |\theta.og| \times |\theta.int|)$$

$$Tot_S =$$
$$(|\theta.sg| \times |\theta.sg| \times |\theta.int|) +$$
$$(|\theta.ag| \times |\theta.ag| \times |\theta.int|) +$$
$$(|\theta.og| \times |\theta.og| \times |\theta.int|)$$

With the totals above defined, the offsetting procedure can be extended to find the indices of both positive and negative facts. The index of a negative fact is offset by Tot_A, the total number of positive facts. For example, given the first negative subject subset fact:

$\neg subst(\theta.sg[0], \theta.sg[0], \theta.int[0])$

The index of the above fact is:

$$Tot_A + Tot_H + Tot_M$$

So far, we have seen that this technique is effective in encoding positive and negative facts in one state only. However, this procedure can be further extended to assign indices to facts in different states. First, we assume that the integers 0 to ($2 \times Tot_A$) - 1 maps to positive and negative facts of state S_0. We can then assign integers ($2 \times Tot_A$) to ($4 \times Tot_A$) - 1 to the positive and negative facts of state S_1. In general, facts of state σ are offset by the facts of state σ - 1.

Algorithm 5.7 shows the implementation of the $Encode()$ function using this technique. The first thing the that the function does is to calculate the offset of the atom α based on the given state σ and the truth indicator τ. The function then takes the ordinal index of each entity and interval in the atom from the symbol table θ. Depending on the atom type, the $Encode()$ then uses one of $Offset\_Holds()$, $Offset\_Memb()$ or $Offset\_Subst()$ with the indices to calculate the appropriate offsets.

Algorithm 5.7 *Encode()*

```
FUNCTION Encode(α, σ, τ)
  offset = σ · Tot_A · 2
  IF τ == false THEN
    offset = offset + Tot_A
  ENDIF
  CASE α.type OF
    holds :
      s = θ_s.index_of(α.holds.sub)
      a = θ_a.index_of(α.holds.acc)
      o = θ_o.index_of(α.holds.obj)
      i = θ_int.index_of(α.holds.int)
      RETURN offset + Offset_Holds(s, a, o, i)
    member :
      CASE α.memb.type OF
        sub :
          e = θ_ss.index_of(α.memb.elt)
          g = θ_sg.index_of(α.memb.grp)
        acc :
          e = θ_as.index_of(α.memb.elt)
          g = θ_ag.index_of(α.memb.grp)
        obj :
          e = θ_os.index_of(α.memb.elt)
          g = θ_og.index_of(α.memb.grp)
      ENDCASE
      i = θ_int.index_of(α.memb.int)
      RETURN offset + Offset_Memb(e, g, i, α.memb.type)
```

```
    subset :
      CASE α.subst.type OF
        sub :
          g0 = θsg.index_of(α.subst.grp0)
          g1 = θsg.index_of(α.subst.grp1)
        acc :
          g0 = θag.index_of(α.subst.grp0)
          g1 = θag.index_of(α.subst.grp1)
        obj :
          g0 = θag.index_of(α.subst.grp0)
          g1 = θag.index_of(α.subst.grp1)
      ENDCASE
      i = θint.index_of(α.subst.int)
      RETURN offset + Offset_Subst(g0, g1, i, α.subst.type)
  ENDCASE
ENDFUNCTION
```

The $Offset\_Holds()$ function shown in Algorithm 5.8 calculates the offset for atoms of type holds by using the given indices of its elements. As discussed above, the subject offset of a holds atom is obtained by multiplying the actual subject index sub by the total number of access rights by the total number of objects by the total number of intervals defined in the symbol table. Similarly, the access right offset is calculated by multiplying the actual access right index acc by the remaining elements: the total number of objects and the total number of intervals. The offsets for objects and intervals are calculated in the same manner. Finally, the sum of these offsets is the offset for the holds atom, which is returned by this function.

Algorithm 5.8 *Offset_Holds()*

```
FUNCTION Offset_Holds(sub, acc, obj, int)
  RETURN
    (sub · (|θ.as| + |θ.ag|) · (|θ.os| + |θ.og|) · |θ.int|) +
    (acc · (|θ.os| + |θ.og|) · |θ.int|) +
    (obj · |θ.int|) +
    int
ENDFUNCTION
```

The calculation of the offset for member atoms as shown in Algorithm 5.9 uses a similar method, except each type (subject, access right and object) is handled in a different way. Firstly, the access right index is offset by the total number of subject member atoms ($|\theta.ss| \cdot |\theta.sg| \cdot |\theta.int|$). The index for member atoms of type object are similarly offset by the total

number of subject and access right member atoms ($|\theta.ss| \cdot |\theta.sg| \cdot |\theta.int| + |\theta.as| \cdot |\theta.ag| \cdot |\theta.int|$). Furthermore, the index of all atoms of type member is offset by the total number of holds atoms (Tot_H).

Algorithm 5.9 *Offset_Memb()*

```
FUNCTION Offset_Memb(elt, grp, int, type)
  CASE type OF
    sub :
      offset =
        (elt · |θ.sg| · |θ.int|) +
        (grp · |θ.int|) +
        int
    acc :
      offset =
        (|θ.ss| · |θ.sg| · |θ.int|) +
        (elt · |θ.ag| · |θ.int|) +
        (grp · |θ.int|) +
        int
    obj :
      offset =
        (|θ.ss| · |θ.sg| · |θ.int|) +
        (|θ.as| · |θ.ag| · |θ.int|) +
        (elt · |θ.og| · |θ.int|) +
        (grp · |θ.int|) +
        int
  ENDCASE

  RETURN Tot_H + offset
ENDFUNCTION
```

Algorithm 5.10 shows how the $Offset\_Subst()$ function calculates the offset for subset atoms. The method is similar to that used by the $Offset\_Memb()$ function, except the final index is offset by the total number of holds and member atoms.

Algorithm 5.10 *Offset_Subst()*

```
FUNCTION Off_Subst(grp0, grp1, int, type)
  CASE type OF
    sub :
      offset =
        (grp0 · |θ.sg| · |θ.int|) +
        (grp1 · |θ.int|) +
```

```
        int
    acc :
      offset =
        (|θ.sg|^2 · |θ.int|) +
        (grp_0 · |θ.ag| · |θ.int|) +
        (grp_1 · |θ.int|) +
        int
    obj :
      offset =
        (|θ.sg|^2 · |θ.int|) +
        (|θ.ag|^2 · |θ.int|) +
        (grp_0 · |θ.og| · |θ.int|) +
        (grp_1 · |θ.int|) +
        int
  ENDCASE

  RETURN Tot_H + Tot_M + offset
ENDFUNCTION
```

5.3.3 Populating the Policy Base

Now, we describe the process of storing the different components of the policy into the policy base structures shown in Section 5.3.1. In this section, this process is broken up into several policy base operators that the policy parser can call as it parses a language $\mathcal{L}^{\mathcal{T}}$ policy.

Registering Entities and Intervals

Before the policy base can be made to perform any task, it must first be made aware of all entities and intervals defined in the policy. The $PB.AddEntity()$, $PB.AddInterval()$ and $PB.AddIntervalEP()$ operators are called by the parser as it goes through the entity and interval declarations of the policy.

The $PB.AddEntity(id, type)$ function shown in Algorithm 5.11 registers the identifier id of type $type$ to the appropriate list in the symbol table.

Algorithm 5.11 *PB.AddEntity()*

```
FUNCTION PB.AddEntity(id, type)
  CASE type OF
    sub-sin :
      PB.symtab.ss.Add(id)
```

```
    sub-grp :
      PB.symtab.sg.Add(id)
    acc-sin :
      PB.symtab.as.Add(id)
    acc-grp :
      PB.symtab.ag.Add(id)
    obj-sin :
      PB.symtab.os.Add(id)
    obj-grp :
      PB.symtab.og.Add(id)
  ENDCASE
ENDFUNCTION
```

Algorithm 5.12 shows the $PB.AddInterval(id)$ operator which registers the interval identifier id to the interval list in the symbol table and the interval network object. While reading the policy, if the parser encounters a declaration of a well-defined interval, it makes a call to the $PB.AddIntervalEP()$ function (Algorithm 5.13) instead. This function registers the interval identifier to both the symbol table and the network object, then binds the interval with the given end points in the network object.

Algorithm 5.12 *PB.AddInterval()*

```
FUNCTION PB.AddInterval(id)
  PB.symtab.int.Add(id);
  PB.netwk.AddInt(id);
ENDFUNCTION
```

Algorithm 5.13 *PB.AddIntervalEP()*

```
FUNCTION PB.AddIntervalEP(id, ep0, ep1)
  PB.AddInterval(id)
  PB.netwk.Bind(id, ep0, ep1)
ENDFUNCTION
```

Adding Temporal Interval Constraints

When the parser encounters a temporal constraint, the $PB.AddRelation()$ operator shown in Algorithm 5.14 is called. The operator normalises the given interval expression first before passing it on to the network object one relation at a time. Note that as shown in Algorithm 5.4, the effects of each added relation is propagated to the entire network.

Algorithm 5.14 *PB.AddRelation()*

```
FUNCTION PB.AddRelation(rexp)
  rlist = NormaliseExp(rexp)
  FOR i = 0 TO (|rlist| - 1) DO
    PB.netwk.AddRel(rlist[i].int0, rlist[i].int1, rlist[i].rs)
  ENDDO
ENDFUNCTION
```

Adding Authorisation Statements

The operators shown in Algorithms 5.15, 5.16 and 5.17 are called by the parser to register initial state rules, authorisation constraint rules and policy update declarations, respectively. While the operator $PB.AddInitially()$ simply adds the initial expression to the initial facts table one fact at a time, the $PB.AddConstraint()$ and $PB.AddUpdate()$ operators add their entries directly into the appropriate table in the policy base.

Algorithm 5.15 *PB.AddInitially()*

```
FUNCTION PB.AddInitially(exp)
  FOR i = 0 TO (|exp| - 1) DO
    PB.inittab.Add(exp[i])
  ENDDO
ENDFUNCTION
```

Algorithm 5.16 *PB.AddConstraint()*

```
FUNCTION PB.AddConstraint(exp0, exp1, exp2, rexp)
  cnode = (exp0, exp1, exp2, NormaliseExp(rexp))
  PB.consttab.Add(cnode)
ENDFUNCTION
```

Algorithm 5.17 *PB.AddUpdate()*

```
FUNCTION PB.AddUpdate(u, exp0, exp1, rexp)
  unode = (u, exp0, exp1, NormaliseExp(rexp))
  PB.updatetab.Add(unode)
ENDFUNCTION
```

Policy Update Sequence Manipulation

Algorithms 5.18, 5.19 and 5.20 are the policy base operators that correspond to the *seq add*, *seq del* and *seq list* language $\mathcal{L}^{\mathcal{T}}$ statements, respectively.

Algorithm 5.18 *PB.AddSequence()*

```
FUNCTION PB.AddSequence(u, ilist)
  snode = (u, ilist)
  PB.seqtab.Add(snode)
ENDFUNCTION
```

Algorithm 5.19 *PB.DelSequence()*

```
FUNCTION PB.DelSequence(index)
  PB.seqtab.Del(index)
ENDFUNCTION
```

Algorithm 5.20 *PB.ListSequence()*

```
FUNCTION PB.ListSequence()
  RETURN PB.seqtab
ENDFUNCTION
```

5.3.4 Calculating the Answer Set

Once the policy update sequence table has been populated by the sequence manipulation operators described above, the policy base must be made to generate answer sets based on the stored policies and updates. The answer sets must be generated prior to any query evaluation requests. This process, of course, is triggered by the parser when it encounters the *compute* statement in the language $\mathcal{L}^{\mathcal{T}}$ policy.

The $GenNLP()$ function in Algorithm 3.1 and the rest of the functions in Section 3.2.3 illustrates how a language $\mathcal{L}$ domain is translated into a normal logic program and from answer sets derived from that normal logic program, how queries are evaluated. This section focuses on extending these methods to generate answer sets from language $\mathcal{L}^{\mathcal{T}}$ domains, as well as showing how these techniques are applied to the policy base structures.

Firstly, we need to formally define a stable models object capable of generating answer sets from normal logic programs. Definition 5.3 formalises a suitable object based on the interface provided by the *SModels* system in [47, 59].

Definition 5.3 *An SModels object SM is a stable models implementation with support for the following operations:*

1. *$SM.Init()$ initialises the object. Any previous operations are reset.*
2. *$SM.RuleBegin()$ marks the beginning of a rule.*
3. *$SM.RuleHead(\alpha)$ registers the atom α as the head of the rule.*

4. *$SM.RuleBody(\alpha, \tau)$ registers the atom α, whose negation-as-failure value is τ, as a body of the current rule.*

5. *$SM.RuleEnd()$ marks the ending of the current rule.*

6. *$SM.GetAnswerSets()$ returns a list of answer sets.*

With the stable model object defined, we can now define the policy base operation that generates an answer set based on the policy stored. Algorithm 5.21 shows the policy base operation that extends the original $GenNLP()$ function to generate an answer set.

Algorithm 5.21 *PB.GenAnswerSets()*

```
FUNCTION PB.GenAnswerSets()
  SM.Init()
  TransInitStateRules(PB.inittab, SM)
  TransConstRules(PB.consttab, PB.symtab, PB.seqtab, PB.netwk, SM)
  TransUpdateRules(PB.updatetab, PB.symtab, PB.seqtab, PB.netwk, SM)
  GenInherRules(PB.symtab, PB.seqtab, SM)
  GenTransRules(PB.symtab, PB.seqtab, SM)
  GenInertRules(PB.symtab, PB.seqtab, SM)
  GenRefleRules(PB.symtab, PB.seqtab, SM)
  GenTempoRules(PB.symtab, PB.seqtab, PB.netwk, SM)
  GenConsiRules(PB.symtab, PB.seqtab, SM)
  RETURN SM.GetAnswerSets()
ENDFUNCTION
```

Note that most of the functions called by $PB.GenAnswerSets()$ are similar to the functions defined in Section 3.2.3. These functions require only little modifications to generate a normal logic program from a language $\mathcal{L}^{\mathcal{T}}$ domain. In fact, the only major difference is that the new functions must support the language $\mathcal{L}^{\mathcal{T}}$ atoms that include an additional temporal interval parameter. This means that most of the rule-generating functions are extended simply by the addition of an extra interval loop, which passes intervals to the new $Encode()$ function discussed in Section 5.3.2.

Generating Temporal Rules

Recall that Section 4.4.2 has shown that translating a language $\mathcal{L}^{\mathcal{T}}$ domain to a normal logic program requires the generation of temporal relation rules. The semantics of the language tells us that the purpose of these rules is to assert that an atom that holds in a particular time interval ι must also hold in every other interval ι' that is "within" ι, that is, ι' is *equal*, is

during, *starts* or *finishes* ι. Algorithm 5.22 shows the $GenTempoRules()$ function which generates these rules.

Algorithm 5.22 *GenTempoRules()*

```
FUNCTION GenTempoRules(θ, ψ, NET, SM)
  GenHldsTempoRules(θ, ψ, NET, SM)
  GenMembTempoRules(θ, ψ, NET, SM)
  GenSubsTempoRules(θ, ψ, NET, SM)
ENDFUNCTION
```

The $GenHldsTempoRules()$ function shown in Algorithm 5.23, as the name implies, generates the temporal rules for *holds* atoms. The first loop goes through every policy update state while the next 3 loops goes through all defined entities. The fifth and sixth loop goes through all possible pairs of intervals defined in the symbol table.

Inside these loops, the relation between the interval pair is retrieved from the interval network structure. The if-statement ensures that one interval is "within" the other. The number 101 is the sum of the values of the relations *equals*, *during*, *starts* and *finishes* as defined in Table 5.2. Thus, if the relation between the pair of intervals is a subset of the relation set $\{$*equal*, *during*, *starts*, *finishes*$\}$, the holds temporal rules are generated. A similar method is used for the other two functions $GenMembTempoRules()$ and $GenSubsTempoRules()$.

Algorithm 5.23 *GenHldsTempoRules()*

```
FUNCTION GenHldsTempoRules(θ, ψ, NET, SM)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|θ.s| - 1) DO
      FOR k = 0 TO (|θ.a| - 1) DO
        FOR l = 0 TO (|θ.o| - 1) DO
          FOR m = 0 TO (|θ.int| - 1) DO
            FOR n = 0 TO (|θ.int| - 1) DO
              rs = NET.get(m, n)
              IF (rs | 101) == 101 THEN
                α0 = {θ.s[j], θ.a[k], θ.o[l], θ.int[n]}
                α1 = {θ.s[j], θ.a[k], θ.o[l], θ.int[m]}
                SM.RuleBegin()
                SM.RuleHead(Encode(α0, i, true))
                SM.RuleBody(Encode(α1, i, true), true)
                SM.RuleEnd()
                SM.RuleBegin()
                SM.RuleHead(Encode(α0, i, false))
                SM.RuleBody(Encode(α1, i, false), true)
```

```
                SM.RuleEnd()
              ENDIF
            ENDDO
          ENDDO
        ENDDO
      ENDDO
    ENDDO
ENDFUNCTION
```

Translating Constraint and Policy Update Rules

The language $\mathcal{L}$ versions of the $TransConstRules()$ and $TransUpdateRules()$ functions defined in Algorithm 3.3 and Algorithm 3.4 assume that expressions are already grounded. In this section, we extend these functions to handle the interval relations of language $\mathcal{L}^{\mathcal{T}}$ as well as ground variables.

First, we define the following functions to be used to ground the variables:

- $GetVList(exp)$ returns a list of variable identifiers that occur in the given expression exp.
- $GenTupleList(vlist)$ returns a list of tuples, given a list of variables $vlist$. Each tuple in the returned list contains entities and intervals of the same type as the corresponding variable in $vlist$.
- $CheckTuple(tuple, rlist, NET)$ returns *true* if the intervals in $tuple$ satisfy the interval relations specified in $rlist$.
- $Replace(vlist, tuple, exp)$. Given a list of variables $vlist$, a list of entities and intervals $tuple$ and an expression exp, the function returns the expression ϵ with all variables in $vlist$ replaced by corresponding entities and intervals from $tuple$.

The $TransConstRules()$ function shown in Algorithm 5.24 starts by creating a list of variables $vlist$ that occurs in the constraint expressions. With this list, a list of tuples $tlist$ is generated by the use of the $GenTupleList()$ function. As the algorithm goes through each tuple in $tlist$, it checks whether the intervals in the current tuple satisfies the temporal constraints associated with the current constraint rule. If the tuple is valid, the algorithm then uses this to ground all the variables in the expressions. Once grounded, the expressions are passed to the SM object to construct the normal logic program rule.

Algorithm 5.24 *TransConstRules()*

```
FUNCTION TransConstRules(ω, θ, ψ, NET, SM)
  FOR i = 0 TO (|ω| - 1) DO
    vlist.Init()
    vlist.Append(GetVList(ω[i].exp)
    vlist.Append(GetVList(ω[i].pcn)
    vlist.Append(GetVList(ω[i].ncn)
    tlist = GenTupleList(θ, vlist)
    rlist = NormaliseExp(ω[i].rexp)
    FOR j = 0 TO (|tlist| - 1) DO
      IF CheckTuple(tlist[j], rlist, NET) THEN
        ε0 = Replace(vlist, tlist[j], ω[i].exp)
        ε1 = Replace(vlist, tlist[j], ω[i].pcn)
        ε2 = Replace(vlist, tlist[j], ω[i].ncn)
        FOR k = 0 TO (|θ| - 1) DO
          FOR l = 0 TO (|ε0| - 1) DO
            SM.RuleBegin()
            SM.RuleHead(Encode(ε0[l].atom, k, ε0[l].truth))
            FOR m = 0 TO (|ε1| - 1) DO
              SM.RuleBody(Encode(ε1[m].atom, k, ε1[m].truth), true)
            ENDDO
            FOR n = 0 TO (|ε2| - 1) DO
              SM.RuleBody(Encode(ε2[n].atom, k, ε2[n].truth), false)
            ENDDO
            SM.RuleEnd()
          ENDDO
        ENDDO
      ENDIF
    ENDDO
  ENDDO
ENDFUNCTION
```

A similar method is used by the $TransUpdateRules()$ function shown in Algorithm 5.25.

Algorithm 5.25 *TransUpdateRules()*

```
FUNCTION TransUpdateRules(ω, θ, ψ, NET, SM)
  FOR i = 0 TO (|ψ| - 1) DO
    FOR j = 0 TO (|ω| - 1) DO
      IF ψ[i].name == ω[j].name THEN
        upd = GndUpdate(ω[j], ψ[i].ilist)
        vlist.Init()
```

```
        vlist.Append(GetVList(upd.pre)
        vlist.Append(GetVList(upd.pst)
        tlist = GenTupleList(θ, vlist)
        rlist = NormaliseExp(ω[j].rexp)
        FOR k = 0 TO (|tlist| - 1) DO
          IF CheckTuple(tlist[k], rlist, NET) THEN
            ϵ0 = Replace(vlist, tlist[k], upd.pre)
            ϵ1 = Replace(vlist, tlist[k], upd.pst)
            FOR l = 0 TO (|ϵ1| - 1) DO
              SM.RuleBegin()
              SM.RuleHead(Encode(ϵ1[l].atom, i + 1, ϵ1[l].truth))
              FOR m = 0 TO (|ϵ0| - 1) DO
                SM.RuleBody(Encode(ϵ0[l].atom, i, ϵ0[l].truth), true)
              ENDDO
              SM.RuleEnd()
            ENDDO
          ENDIF
        ENDDO
      ENDIF
    ENDDO
  ENDDO
ENDFUNCTION
```

5.3.5 Evaluating Query Expressions

Once the answer sets have been generated, the policy base can accept query evaluation requests through the $PB.EvaluateExp(exp)$ operator. The operator works by checking the presence of queried facts or their negations in every answer sets generated. Algorithm 5.26 shows how a given expression exp can be evaluated against a list of answer sets Λ, as returned by the $PB.GenAnswerSets()$ operator.

Algorithm 5.26 *PB.EvaluateExp()*

```
FUNCTION PB.EvaluateExp(exp, Λ)
  result = true
  FOR i = 0 TO (|exp| - 1) DO
    rv = EvaluateFact(exp[i], |PB.seqtab|, Λ)
    IF rv == false THEN
      RETURN false
    ELSE IF rv == unknown THEN
      result = unknown
    ENDIF
```

```
  ENDDO
  RETURN result
ENDFUNCTION
```

The $PB.EvaluateExp()$ operator evaluates each fact from the given query expression by calling the function $EvaluateFact()$ shown in Algorithm 5.27 which evaluates a single fact ρ in state σ, against a list of answer sets Λ.

Algorithm 5.27 *EvaluateFact()*

```
FUNCTION EvaluateFact(ρ, σ, Λ)
  IF IsFactIn(Encode(ρ.atom, σ, ρ.truth), Λ) THEN
    RETURN true
  ELSE IF IsFactIn(Encode(ρ.atom, σ, NOT ρ.truth), Λ) THEN
    RETURN false
  ENDIF
  RETURN unknown
ENDFUNCTION
```

The function $IsFactIn()$ simply returns a boolean value to indicate whether or not the given fact index $index$, as returned by the $Encode()$ function, is present in every answer set in Λ.

Algorithm 5.28 *IsFactIn()*

```
FUNCTION IsFactIn(index, Λ)
  FOR i = 0 TO (|Λ| - 1) DO
    IF NOT Λ[i].Find(index) THEN
      RETURN false
    ENDIF
  ENDDO
  RETURN true
ENDFUNCTION
```

5.4 Experimental Analysis and Discussions

So far, the chapter has shown the full implementation details of two systems. First, we described a simple but efficient implementation of a temporal interval reasoner engine based on Allen's interval algebra and propagation algorithm [2]. Secondly, the chapter discussed in detail the algorithms and data structures that make up the extended PolicyUpdater authorisation system from an implementation point of view.

In this section, we focus our attention to the performance of the implementation by comparing the computation times of the system against different domain sizes. Like the tests described in Section 3.3, these tests were conducted on an AMD Athlon XP 1800+ machine with 1GB of RAM, running Debian GNU/Linux 3.1 with a plain Linux 2.6.16.20 kernel. These tests are based on the latest version of PolicyUpdater 2 (Vlad 2.0.1) compiled with Tribe 0.4.0 and SModels 2.31.

Table 5.12 shows the domain sizes used for each test case. S_{E_s} and S_{E_g} are the numbers of singular and group entities, respectively; S_ι is the number of intervals; S_I is the number of initial state facts; S_{C_a} and S_{C_ι} denotes the number of authorisation and interval constraint rules, respectively; 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_ι	S_I	S_{C_a}	S_{C_ι}	S_U	S_S	S_Q
1	4	3	3	3	1	3	1	1	4
2	24	23	3	3	1	3	1	1	4
3	104	3	3	3	1	3	1	1	4
4	4	103	3	3	1	3	1	1	4
5	24	23	3	103	1	3	1	1	4
6	24	23	3	3	101	3	1	1	4
7	24	23	3	3	1	3	101	1	4
8	24	23	3	3	1	3	101	101	4
9	24	23	3	3	1	3	1	1	104
10	24	23	3	103	1	3	101	101	4
11	24	23	3	3	101	3	101	101	4
12	24	23	3	103	101	3	101	101	104
13	104	103	3	103	101	3	101	101	104
14	24	23	103	3	1	3	1	1	4
15	24	23	23	3	1	103	1	1	4
16	24	23	103	3	1	103	1	1	4
17	104	103	103	103	101	103	101	101	104

Table 5.12: Seventeen Test Cases with Different Domain Sizes

The tests were conducted using the code listing in Example 4.2 as the base case (test case 1). For all other test cases, entities, intervals, rules and/or queries were added or removed to match the domain sizes shown in Table 5.12. Note that the language $\mathcal{L}^{\mathcal{T}}$ test program used in these tests is similar to the language $\mathcal{L}$ test program used in the tests described in Section 3.3. In fact, the first 13 test cases were chosen to match those used to test PolicyUpdater 1.

Each test case was executed 10 times. Table 5.13 shows the average execution times in seconds for each test case. T_C is the total time spent by the system to compute the answer

	T_C	T_Q
1	0.009178	0.002628
2	2.645432	2.972180
3	0.665680	0.904036
4	142.137921	141.881564
5	2.658151	3.002972
6	2.671187	2.997832
7	2.652458	2.955020
8	143.035015	154.559300
9	2.660288	80.922608
10	143.035966	156.002920
11	143.105090	155.951128
12	143.341907	1937.619320
13	-	-
14	91.684118	117.606152
15	20.186534	24.214144
16	92.138305	117.525392
17	-	-

Table 5.13: Average Computation Times in Seconds (PolicyUpdater 2)

sets, while T_Q is the total time used by the system to evaluate all the queries. In test cases 13 and 17, the test system's memory was exhausted before the tests were completed. This is due to the very large number of rules generated from the test program.

The overall conclusion that can be drawn from these test results is that PolicyUpdater 2's execution times are much higher, as compared to that of PolicyUpdater 1 (shown in Table 3.8). This can be explained by the increase in the number of atoms. Recall that the main difference between language $\mathcal{L}$ and language $\mathcal{L}^{\mathcal{T}}$ is that the latter includes an interval attribute in its atom definition. The consequence of this is that the number of atoms in the domain increases significantly as the number of intervals increases. Of course, as the number of atoms increases, so does the size of the answer sets, which in turn means greater execution times. To support this argument, we compare test cases 14 and 15. In test case 14, the number of intervals was increased to 103 while the number of interval constraints were left at 3. In test case 15, the number of interval constraints was increased only to 23 but the interval constraints were increased to 103. The results show that increasing the number of intervals has a more profound effect on computation and query times than increasing the interval constraints.

The algorithms and data structures discussed in this chapter were designed as a compromise between efficiency and simplicity. Although certain algorithms used by the system

can be optimised for speed or size, these optimisations would introduce another level of complexity that fall beyond the scope of this thesis.

Chapter 6

Conclusion

Authorisation or access control is an important part of information security systems. Although several access control approaches and models have been proposed and used over the years, very few maintain a good balance between flexibility, expressiveness and implementation. That is, those access control systems with simple implementations often lack the ability to express complex authorisation rules or the flexibility required for policy updates. On the other hand, those models that can handle policy updates, conditional rules or temporal constraints often lack the details necessary for full system implementation.

For example, several systems, such as those proposed by Jajodia et al. [32, 33, 34] and Bertino et al. [13] provide flexible logic-based approaches to access control, but they do not address implementation issues. In contrast, the scheme proposed by Ray [49] focuses on the implementation of an authorisation system with support for policy updates, but it does not support negative authorisations and conditional rules. Several authorisation models like [11, 12, 53] allow authorisation rules to have temporal properties expressed as time intervals. However, these systems lack the means to express the relationships between these time intervals themselves.

In this thesis, we have presented not only a logic-based authorisation model, but the implementation details of a full authorisation system with support for policy updates and temporal interval relations.

The PolicyUpdater system is an authorisation system that uses a first-order logic language, language $\mathcal{L}$, to represent the authorisation policy or policy base. A language $\mathcal{L}$ atom, composed of a subject, an access right and an object, is used to construct authorisation facts and rules, which in turn defines the policy base. Through the use of this language, the PolicyUpdater system is capable of expressing and evaluating both positive and negative authorisations from policies with conditional logic rules. The two key features of this language, and therefore the system, is its ability to express default authorisations in conditional rules and conditional policy update rules which can be applied in sequence.

We have also shown from the semantics of language $\mathcal{L}$ that from a given policy expressed in this language and a sequence of policy updates, an extended logic program can be generated, which in turn can be translated into a normal logic program. In our implementation, we have shown that through these translations and the stable model semantics, a set of answer sets can be generated from any consistent language $\mathcal{L}$ policy. The PolicyUpdater system uses these answer sets to evaluate authorisation queries.

In Chapter 3, we have outlined the data structures and algorithms that make up the PolicyUpdater system. The performance analysis of the system shows that, given a realistic input size of policies and queries, our implementation performs reasonably well in terms of computation speed. In the case study, we have shown an application in which the PolicyUpdater system is used as an authorisation system for a web server.

The extended version of the system, PolicyUpdater 2, addressed the issue of expressing temporal constraints in the authorisation policy. This is made possible by the integration of the well-established temporal interval algebra into a logic-based authorisation language, language $\mathcal{L}^{\mathcal{T}}$. This non-trivial extension of language $\mathcal{L}$ redefines the authorisation atom to include a time interval within which the authorisation is valid. More importantly, the interval algebra gives the language the ability to express relations between the time intervals themselves, thereby permitting the expression of rules that contain these interval relations. For example, language $\mathcal{LT}$ allows rules such as "If $Alice$ is allowed to $read$ file f at interval i_0, then Bob should be allowed to $read$ file f at interval i_1, where i_1 is during i_0". As with language $\mathcal{L}$, we have shown how language $\mathcal{L}^{\mathcal{T}}$ policies can be translated into normal logic programs for evaluation using the stable model semantics.

Like the first version, the full implementation details of the extended version of PolicyUpdater were shown in Chapter 5, including the implementation of a separate temporal interval relation reasoning engine. The chapter also included the details of the integration of this engine with the rest of the authorisation system.

To the best of our knowledge, PolicyUpdater is the first fully implemented logic-based authorisation system that is capable of expressing default authorisation rules and has support for dynamic and conditional policy updates. Similarly, we believe that PolicyUpdater 2 is the first logic-based authorisation system with the same capabilities as version 1, and with the ability to support temporal constraints expressed in the interval relation algebra.

At the time of writing, the latest version of the PolicyUpdater 2 implementation is Vlad 2.0.1. This version includes full variable grounding, dynamic policy updates and temporal constraints support. The latest version of the PolicyUpdater 1 implementation is Vlad 1.4.3. Since the PolicyUpdater source code is over 14,000 lines long, the code listing is not included in this thesis. However, the source code for both versions is available for download from the project website:

http://www.scm.uws.edu.au/~jcrescin/projects/policyupdater/index.html

As for future work, there are several possible directions worth pursuing. On the theoretical side, we note that the PolicyUpdater system handles conflicts in authorisation rules by giving negative authorisations a higher precedence over positive ones. Although our tests shows that this conservative discipline is sufficiently safe for most applications, other applications or environments may require a more fine-tuned conflict resolution strategy. One such strategy might be to incorporate the expression of prioritised authorisation rules into the authorisation language. This way, a conflict is resolved by giving a higher precedence to rules with higher priorities. Another possible future work direction is to extend the system to allow disjunctive information to be used in authorisation rules.

On the implementation side, we note that testing a temporal interval relation network for consistency is an intractable problem. Although our tests with small networks have shown negligible effects on computation time, the effects will be a problem for networks with thousands of interval relations. One possible future work is to replace the system's support for the full interval algebra with one of its tractable sub-algebras [35] while maintaining equivalent, or near equivalent expressive power.

Appendix A

Language Specification

A.1 Language $\mathcal{L}$ in Backus-Naur Form

```
<start>                 : <program>

<program>               : <head> <body> <tail>

<head>                  : <nil>

<body>                  : <entity-section>
                          <initial-section>
                          <constraint-section>
                          <update-section>
                          <directive-section>

<tail>                  : <nil>

<entity-section>        : <nil> |
                          <entity-stmt-list>

<initial-section>       : <nil> |
                          <inital-stmt-list>

<constraint-section>    : <nil> |
                          <constraint-stmt-list>

<update-section>        : <nil> |
```

```
                              <update-stmt-list>

<directive-section>       : <nil> |
                              <directive-stmt-list>

<entity-stmt-list>        : <entity-stmt> |
                              <entity-stmt-list>
                              <entity-stmt>

<initial-stmt-list>       : <initial-stmt> |
                              <initial-stmt-list>
                              <initial-stmt>

<constraint-stmt-list>    : <constraint-stmt> |
                              <constraint-stmt-list>
                              <constraint-stmt>

<update-stmt-list>        : <update-stmt> |
                              <update-stmt-list>
                              <update-stmt>

<directive-stmt-list>     : <directive-stmt> |
                              <directive-stmt-list>
                              <directive-stmt>

<entity-stmt>             : <entity>
                              <entity-declaration>
                              <semicolon>

<entity-declaration>      : <sub-entity-decl> |
                              <obj-entity-decl> |
                              <acc-entity-decl> |
                              <sub-grp-entity-decl> |
                              <obj-grp-entity-decl> |
                              <acc-grp-entity-decl>

<sub-entity-decl>         : <sub-type>
```

```
                          <sub-entity-list>

<acc-entity-decl>       : <acc-type>
                          <acc-entity-list>

<obj-entity-decl>       : <obj-type>
                          <obj-entity-list>

<sub-grp-entity-decl>   : <sub-grp-type>
                          <sub-grp-entity-list>

<acc-grp-entity-decl>   : <acc-grp-type>
                          <acc-grp-entity-list>

<obj-grp-entity-decl>   : <obj-grp-type>
                          <obj-grp-entity-list>

<sub-entity-list>       : <identifier> |
                          <sub-entity-list> <comma>
                          <identifier>

<obj-entity-list>       : <identifier> |
                          <obj-entity-list> <comma>
                          <identifier>

<acc-entity-list>       : <identifier> |
                          <acc-entity-list> <comma>
                          <identifier>

<sub-grp-entity-list>   : <identifier> |
                          <sub-grp-entity-list>
                          <comma> <identifier>

<obj-grp-entity-list>   : <identifier> |
                          <obj-grp-entity-list>
                          <comma> <identifier>
```

```
<acc-grp-entity-list>    : <identifier> |
                           <acc-grp-entity-list>
                           <comma> <identifier>

<initial-stmt>           : <initially> <expression>
                           <semicolon>

<constraint-stmt>        : <always> <expression>
                           <implied-clause>
                           <with-clause> <semicolon>

<implied-clause>         : <nil> |
                           <implied> <by> <expression>

<with-clause>            : <nil> |
                           <with> <absence>
                           <expression>

<update-stmt>            : <identifier>
                           <update-var-def> <causes>
                           <expression> <if-clause>
                           <semicolon>

<if-clause>              : <nil> |
                           <if> <expression>

<update-var-def>         : <open-parent>
                           <close-parent> |
                           <open-parent>
                           <update-var-list>
                           <close-parent>

<update-var-list>        : <identifier> |
                           <update-var-list>
                           <comma> <identifier>

<directive-stmt>         : <sequence-stmt> |
```

```
                          <compute-stmt> |
                          <query-stmt>

<sequence-stmt>         : <sequence>
                          <sequence-cmd-clause>
                          <semicolon>

<compute-stmt>          : <compute> <semicolon>

<query-stmt>            : <query> <expression>
                          <semicolon>

<sequence-cmd-clause>   : <sequence-add-clause> |
                          <sequence-del-clause> |
                          <sequence-lst-clause>

<sequence-add-clause>   : <add> <update-ref-def>

<sequence-del-clause>   : <del> <number>

<sequence-lst-clause>   : <list>

<update-ref-def>        : <identifier> <open-parent>
                          <update-ref-ident-args>
                          <close-parent>

<update-ref-ident-args> : <nil> |
                          <update-ref-ident-list>

<update-ref-ident-list> : <identifier> |
                          <update-ref-ident-list>
                          <comma> <identifier>

<expression>            : <expression> <logical-op>
                          <boolean-fact> |
                          <boolean-fact>
```

```
<boolean-fact>          : <not> <fact> |
                          <fact>

<fact>                  : <holds-fact> |
                          <subst-fact> |
                          <memb-fact>

<holds-fact>            : <holds> <open-parent>
                          <identifier> <comma>
                          <identifier> <comma>
                          <identifier> <close-parent>

<subst-fact>            : <subset> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<memb-fact>             : <member> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<logical-op>            : <and>

<identifier>            : <alpha> <alphanum>

<and>                   : <comma>

<not>                   : !

<holds>                 : holds

<member>                : memb

<subset>                : subst

<initially>             : initially

<causes>                : causes
```

```
<implied>                : implied

<by>                     : by

<with>                   : with

<absence>                : absence

<always>                 : always

<if>                     : if

<query>                  : query

<compute>                : compute

<sequence>               : seq

<entity>                 : entity

<sub-type>               : sub

<obj-type>               : obj

<acc-type>               : acc

<sub-grp-type>           : sub-grp

<obj-grp-type>           : obj-grp

<acc-grp-type>           : acc-grp

<add>                    : add

<del>                    : del
```

```
<list>                  : list

<open-parent>           : (

<close-parent>          : )

<comma>                 : ,

<semicolon>             : ;

<underscore>            : _

<digit>                 : [0-9]

<number>                : <digit> |
                          <number> <digit>

<alpha>                 : [a-zA-Z]

<alphanum>              : <alpha> |
                          <digit> |
                          <underscore>
```

A.2 Language $\mathcal{L}^{\mathcal{T}}$ in Backus-Naur Form

```
<start>                 : <program>

<program>               : <head> <body> <tail>

<head>                  : <nil>

<body>                  : <identifier-section>
                          <initial-section>
                          <relation-section>
                          <constraint-section>
                          <update-section>
                          <directive-section>
```

```
<tail>                    : <nil>

<identifier-section>      : <nil> |
                            <identifier-stmt-list>

<initial-section>         : <nil> |
                            <inital-stmt-list>

<relation-section>        : <nil> |
                            <relation-stmt-list>

<constraint-section>      : <nil> |
                            <constraint-stmt-list>

<update-section>          : <nil> |
                            <update-stmt-list>

<directive-section>       : <nil> |
                            <directive-stmt-list>

<identifier-stmt-list>    : <identifier-stmt> |
                            <identifier-stmt-list>
                            <identifier-stmt>

<initial-stmt-list>       : <initial-stmt> |
                            <initial-stmt-list>
                            <initial-stmt>

<relation-stmt-list>      : <relation-stmt> |
                            <relation-stmt-list>
                            <relation-stmt>

<constraint-stmt-list>    : <constraint-stmt> |
                            <constraint-stmt-list>
                            <constraint-stmt>
```

```
<update-stmt-list>       : <update-stmt> |
                           <update-stmt-list>
                           <update-stmt>

<directive-stmt-list>    : <directive-stmt> |
                           <directive-stmt-list>
                           <directive-stmt>

<identifier-stmt>        : <entity>
                           <entity-declaration>
                           <semicolon> |
                           <interval>
                           <interval-declaration>
                           <semicolon>

<entity-declaration>     : <sub-entity-decl> |
                           <obj-entity-decl> |
                           <acc-entity-decl> |
                           <sub-grp-entity-decl> |
                           <obj-grp-entity-decl> |
                           <acc-grp-entity-decl>

<interval-declaration>   : <interval-decl> |
                           <interval-declaration>
                           <comma> <interval-decl>

<sub-entity-decl>        : <sub-type>
                           <sub-entity-list>

<acc-entity-decl>        : <acc-type>
                           <acc-entity-list>

<obj-entity-decl>        : <obj-type>
                           <obj-entity-list>

<sub-grp-entity-decl>    : <sub-grp-type>
                           <sub-grp-entity-list>
```

```
<acc-grp-entity-decl>     : <acc-grp-type>
                            <acc-grp-entity-list>

<obj-grp-entity-decl>     : <obj-grp-type>
                            <obj-grp-entity-list>

<sub-entity-list>         : <identifier> |
                            <sub-entity-list> <comma>
                            <identifier>

<obj-entity-list>         : <identifier> |
                            <obj-entity-list> <comma>
                            <identifier>

<acc-entity-list>         : <identifier> |
                            <acc-entity-list> <comma>
                            <identifier>

<sub-grp-entity-list>     : <identifier> |
                            <sub-grp-entity-list>
                            <comma> <identifier>

<obj-grp-entity-list>     : <identifier> |
                            <obj-grp-entity-list>
                            <comma> <identifier>

<acc-grp-entity-list>     : <identifier> |
                            <acc-grp-entity-list>
                            <comma> <identifier>

<interval-decl>           : <identifier>
                            <interval-enpoint-decl> |
                            <identifier>

<interval-endpoint-decl> : <open-bracket> <number>
                            <comma> <number>
```

```
                          <close-braket>

<initial-stmt>          : <initially> <expression>
                          <semicolon>

<relation-stmt>         : <relation> <relation-list>
                          <semicolon>

<relation-list>         : <relation-atom> |
                          <relation-list> <comma>
                          <relation-atom>

<relation-atom>         : <rel-eql-atom> |
                          <rel-bef-atom> |
                          <rel-dur-atom> |
                          <rel-ovr-atom> |
                          <rel-met-atom> |
                          <rel-sta-atom> |
                          <rel-fin-atom>

<rel-eql-atom>          : <eql> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-bef-atom>          : <bef> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-dur-atom>          : <dur> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-ovr-atom>          : <ovr> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-met-atom>          : <met> <open-parent>
```

```
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-sta-atom>          : <sta> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<rel-fin-atom>          : <fin> <open-parent>
                          <identifier> <comma>
                          <identifier> <close-parent>

<constraint-stmt>       : <always> <expression>
                          <implied-clause>
                          <with-clause>
                          <where-clause>
                          <semicolon>

<implied-clause>        : <nil> |
                          <implied> <by> <expression>

<with-clause>           : <nil> |
                          <with> <absence>
                          <expression>

<where-clause>          : <nil> |
                          <where> <relation-list>

<update-stmt>           : <identifier>
                          <update-var-def> <causes>
                          <expression> <if-clause>
                          <where-clause> <semicolon>

<if-clause>             : <nil> |
                          <if> <expression>

<update-var-def>        : <open-parent>
                          <close-parent> |
```

```
                          <open-parent>
                          <update-var-list>
                          <close-parent>

<update-var-list>       : <identifier> |
                          <update-var-list> <comma>
                          <identifier>

<directive-stmt>        : <sequence-stmt> |
                          <compute-stmt> |
                          <query-stmt>

<sequence-stmt>         : <sequence>
                          <sequence-cmd-clause>
                          <semicolon>

<compute-stmt>          : <compute> <semicolon>

<query-stmt>            : <query> <expression>
                          <semicolon>

<sequence-cmd-clause>   : <sequence-add-clause> |
                          <sequence-del-clause> |
                          <sequence-lst-clause>

<sequence-add-clause>   : <add> <update-ref-def>

<sequence-del-clause>   : <del> <number>

<sequence-lst-clause>   : <list>

<update-ref-def>        : <identifier> <open-parent>
                          <update-ref-ident-args>
                          <close-parent>

<update-ref-ident-args> : <nil> |
                          <update-ref-ident-list>
```

```
<update-ref-ident-list>  : <identifier> |
                           <update-ref-ident-list>
                           <comma> <identifier>

<expression>             : <expression> <logical-op>
                           <boolean-fact> |
                           <boolean-fact>

<boolean-fact>           : <not> <fact> |
                           <fact>

<fact>                   : <holds-fact> |
                           <subst-fact> |
                           <memb-fact>

<holds-fact>             : <holds> <open-parent>
                           <identifier> <comma>
                           <identifier> <comma>
                           <identifier> <comma>
                           <identifier> <close-parent>

<subst-fact>             : <subset> <open-parent>
                           <identifier> <comma>
                           <identifier> <comma>
                           <identifier> <close-parent>

<memb-fact>              : <member> <open-parent>
                           <identifier> <comma>
                           <identifier> <comma>
                           <identifier> <close-parent>

<logical-op>             : <and>

<identifier>             : <alpha> <alphanum>

<and>                    : <comma>
```

```
<not>                    : !

<holds>                  : holds

<member>                 : memb

<subset>                 : subst

<initially>              : initially

<relation>               : relation

<causes>                 : causes

<implied>                : implied

<by>                     : by

<with>                   : with

<absence>                : absence

<where>                  : where

<always>                 : always

<if>                     : if

<query>                  : query

<compute>                : compute

<sequence>               : seq

<entity>                 : entity
```

```
<interval>              : interval

<sub-type>              : sub

<obj-type>              : obj

<acc-type>              : acc

<sub-grp-type>          : sub-grp

<obj-grp-type>          : obj-grp

<acc-grp-type>          : acc-grp

<eql>                   : equals

<bef>                   : before

<dur>                   : during

<ovr>                   : overlaps

<met>                   : meets

<sta>                   : starts

<fin>                   : finishes

<add>                   : add

<del>                   : del

<list>                  : list

<open-parent>           : (

<close-parent>          : )
```

```
<open-bracket>          : [

<close-bracket>         : ]

<comma>                 : ,

<semicolon>             : ;

<underscore>            : _

<digit>                 : [0-9]

<number>                : <digit>

<alpha>                 : [a-zA-Z]

<alphanum>              : <alpha> |
                          <digit> |
                          <underscore>
```

Bibliography

[1] Abadi M., Burrows M., Lampson B., Plotkin G., A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 4, pp. 706-734, 1993.

[2] Allen J. F., Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, Vol. 26, No. 11, pp. 832-843, 1983.

[3] Apache Software Foundation, Authentication, Authorization and Access Control. Apache HTTP Server Version 2.1 Documentation, 2004.

`http://httpd.apache.org/docs-2.1/`

[4] Atluri V., Gal A., An Authorization Model for Temporal and Derived Data: Securing Information Portals. *ACM Transactions on Information and System Security*, Vol. 5, No. 1, pp. 62-94, 2002.

[5] Bai Y., Varadharajan V., A Language for Specifying Sequences of Authorization Transformations and Its Applications. In *Proceedings of the First International Conference on Information and Communication Security*, pp. 39-49, 1997.

[6] Bai Y., Varadharajan V., On Transformation of Authorization Policies. *Data and Knowledge Engineering*, Vol. 45, No. 3, pp. 333-357, 2003.

[7] Bai Y., Zhang Y., Varadharajan V., On the Sequence of Authorization Policy Transformations. *International Journal of Information Security*, Vol. 4, No. 1-2, pp. 120-131, 2005.

[8] Baral C., Knowledge, Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, UK. pp. 99-100, 2003.

[9] Bell D. E., LaPadula L. J., Secure Computer Systems: Mathematical Foundations. *Technical Report MTR-2547*, Vol. 1, Mitre Corporation, 1973.

[10] Bell D. E., LaPadula L. J., Secure Computer Systems: Mathematical Foundations and Model. *Technical Report M74-244*, Vol. 1, Mitre Corporation, 1974.

[11] Bertino E., Bettini C., Samarati P., A Temporal Authorization Model. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pp. 26-135, 1994.

[12] Bertino E., Bettini C., Ferrari E., Samarati P., An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, Vol. 23, No. 3, pp. 231-285, 1999.

[13] Bertino E., Buccafurri F., Ferrari E., Rullo P., A Logic-Based Approach for Enforcing Access Control. *Journal of Computer Security*, Vol. 8, No. 2-3, pp. 109-140, 2000.

[14] Bertino E., Mileo A., Provetti A., Policy Monitoring with User-Preferences in PDL. In *Proceedings of IJCAI-03 Workshop for Nonmonotonic Reasoning, Action and Change*, pp. 37-44, 2003.

[15] Bertino E., Mileo A., Provetti A., PDL with Preferences. In *Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pp. 213-222, 2005.

[16] Castano S., Fugini M., Martella G., Samarati P., Database Security. Addison-Wesley Publishing Co., 1995.

[17] Chomicki J., Lobo J., Naqvi S., A Logic Programming Approach to Conflict Resolution in Policy Management. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pp. 121-132, 2000.

[18] Conway R. W., Maxwell W. L., Morgan H. L., On the Implementation of Security Measures in Information Systems. *Communications of the ACM* Vol. 15, No. 4, pp. 211-220, 1972.

[19] Crescini V. F., Zhang Y., A Logic Based Approach for Dynamic Access Control. In *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence (AI 2004, LNCS/LNAI)* Vol. 3339, pp. 623-635, 2004.

[20] Crescini V. F., Zhang Y., PolicyUpdater: A System for Dynamic Access Control. *International Journal of Information Security*, Vol. 5, No. 3, pp. 145-165, 2006.

[21] Crescini V. F., Zhang Y., Expressing Temporal Constraints with the PolicyUpdater System. (to be submitted), 2006.

[22] Crescini V. F., Zhang Y., Wang W., Web Server Authorisation with the PolicyUpdater Access Control System. In *Proceedings of the IADIS International Conference (WWW/Internet 2004)*, Vol. 2, pp. 945-948, 2004.

[23] Farby R., Capability-Based Addressing. *Communications of the ACM*, Vol. 17, No. 7, pp. 403-412, 1974.

[24] Ferraiolo D. F., Kuhn D. R., Role Based Access Controls, In *Proceedings of the 15th National Computer Security Conference*, pp. 554-563, 1992.

[25] Ferraiolo D. F., Cugini J. Kuhn D. R., Role Based Access Control (RBAC): Features and Motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference (CSAC-95)*, pp. 241-248, 1995.

[26] Freuder E. C., A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, Vol. 29, No. 1, pp. 24-32, 1982.

[27] Gelfond M., Lifschitz V., The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pp. 1070-1080, 1988.

[28] Gelfond M., Lifschitz V., Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, Vol. 9, No. 3-4, pp. 365-386, 1991.

[29] Graham G. S., Denning P. J., Protection - Principles and Practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 40, pp. 427-429, 1972.

[30] Halpern J. Y., Weissman V., Using First-Order Logic to Reason About Policies. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pp. 187-201, 2003.

[31] Harrison M. H., Ruzzo W. L., Ullman J. D., Protection in Operating Systems. *Communications of the ACM*, Vol. 19, No. 8, pp. 461-471, 1976.

[32] Jajodia S., Samarati P., Subrahmanian V. S., A Logical Language for Expressing Authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 31-42, 1997.

[33] Jajodia S., Samarati P., Subrahmanian V. S., Bertino E., A Unified Framework for Enforcing Multiple Access Control Policies. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 474-485, 1997.

[34] Jajodia S., Samarati P., Sapino M. L., Subrahmanian V. S., Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, Vol. 26, No. 2, pp. 214-260, 2001.

[35] Krokhin A., Jeavons P., Jonsson P., Reasoning about Temporal Relations: The Tractable Subalgebras of Allen's Interval Algebra. *Journal of the ACM*, Vol. 50, No. 5, pp. 591-640, 2003.

[36] Ladkin P. B., Reinefeld A., Fast Algebraic Methods for Interval Constraint Problems. *Annals of Mathematics and Artificial Intelligence*, Vol. 19, No. 3-4, pp. 383-411, 1997.

[37] Lampson, B. W., Protection. In *Proceedings of the 5th Princeton Symposium on Information Science and Systems*, pp. 437-464, 1971.

[38] Laurie B., Laurie P., Apache: The Definitive Guide (3rd Edition). O'Reilly & Associates Inc., CA, 2003.

[39] Levy, H. M., Capability-Based Computer Systems. DEC Press, 1984.

[40] Li N., Grosof B. N., Feigenbaum J., Delegation Logic: A Logic-Based Approach to Distributed Authorization. *ACM Transactions on Information and System Security (TISSEC)*, Vol. 6, No. 1, pp. 128-171, 2003.

[41] Lin F., Zhao X., On Odd and Even Cycles in Normal Logic Programs. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)* and *16th Conference on Innovative Applications of Artificial Intelligence*, pp. 80, 2004.

[42] Lobo J., Bhatia R., Naqvi S., A Policy Description Language. In *Proceedings of the 16th AAAI National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence*, pp. 291-298, 1999.

[43] Mackworth, A.K., Consistency in Networks of Relations. *Artificial Intelligence*, Vol. 8, No. 1, pp. 99-118, 1977.

[44] Meadows C., Policies for Dynamic Upgrading. *Database Security, IV: Status and Prospects (DBSec)*, pp. 241-250, 1990.

[45] Network Working Group, HTTP 1.1 (RFC 2616). The Internet Society, 1999.
`ftp://ftp.isi.edu/in-notes/rfc2616.txt`

[46] Network Working Group, HTTP Authentication: Basic and Digest Access Authentication (RFC 2617). The Internet Society, 1999.
`ftp://ftp.isi.edu/in-notes/rfc2617.txt`

[47] Niemelä I., Simons P., Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)* and *Lecture Notes in Computer Science*, Vol. 1265, pp. 421-430, 1997.

[48] Organization for the Advancement of Structured Information Standards (OASIS), Extensible Access Control Markup Language (XACML) Specification.

`http://www.oasis-open.org/committees/xacml/`

[49] Ray I., Real-Time Update of Access Control Policies. *Data & Knowledge Engineering*, Vol. 49, No. 3, pp. 287-309, 2004.

[50] Reiter R., A Logic for Default Reasoning. *Artificial Intelligence*, Vol. 13, No. 1-2, pp. 81-132, 1980.

[51] Ruan C., Varadharajan V., Zhang Y., Logic-Based Reasoning on Delegatable Authorizations. In *Proceedings of the 13th International Symposium on Foundations of Intelligent Systems (LNCS)*, Vol. 2366, pp. 185-193, 2002.

[52] Ruan C., Varadharajan V., Zhang Y., Evaluation of Authorization with Delegation and Negation. In *Proceedings of the International Intelligent Information Processing and Web Mining Conference (IIPWM-03)*, pp. 547-551, 2003.

[53] Ruan C., Varadharajan V., Zhang Y., A Logic Model for Temporal Authorization Delegation with Negation. In *Proceedings of the 6th International Conference on Information Security (ISC2003)*, Vol. 2851, pp. 310-324, 2003.

[54] Sandhu R. S., Transformation of Access Rights. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 259-268, 1989.

[55] Sandhu R. S., Coyne E. J., Feinstein H. L., Youman C. E., Role-Based Access Control Models. *IEEE Computer*, Vol. 29, No. 2, pp. 38-47, 1996.

[56] Sandhu R. S., Ganta S., On the Expressive Power of the Unary Transformation Model. In *Proceedings of the Third European Symposium on Research in Computer Security*, pp. 301-318, 1994.

[57] Sandhu R. S., Suri G. S., Non-Monotonic Transformation of Access Rights. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 148-161, 1992.

[58] Simons P., Efficient Implementation of the Stable Model Semantics for Normal Logic Programs. Research Reports Number A35, Helsinki University of Technology, 1995.

`http://www.tcs.hut.fi/Publications/reports/A35.ps.Z`

[59] Simons P., Niemelä I., Soininen T., Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, Vol. 138, No. 1-2, pp. 181-234, 2002.

[60] Valdez-Perez R. E., The Satisfiability of Temporal Constraint Networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 256-260, 1987.

[61] Van Beek P. G., Cohen, R., Exact and Approximate Reasoning about Temporal Relations. *Computational Intelligence*, Vol. 6., No. 3, pp. 132-147, 1990.

[62] Van Beek P. G., Manchak D. W., The Design and Experimental Analysis of Algorithms for Temporal Reasoning. *Journal of Artificial Intelligence Research*, Vol. 4, pp. 1-18, 1996.

[63] Vilain M. B., Kautz H. A., Van Beek P. G., Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report. In *Readings in Qualitative Reasoning about Physical Systems*, pp. 373-381, 1989.

[64] Woo T. Y. C., Lam S. S., Authorization in Distributed Systems: A Formal Approach. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 33-51, 1992.

[65] Woo T. Y. C., Lam S. S., Authorization in Distributed Systems: A New Approach. *Journal of Computer Security*, Vol. 2, No. 2-3, pp. 107-136, 1993.

[66] Zhang Y., Handling Defeasibilities in Action Domains. *Theory and Practice of Logic Programming*, Vol. 3, No. 3, pp. 329-376, 2003.