Journal of Computer Security
IOS Press

AUTHORIZATION IN DISTRIBUTED SYSTEMS: A NEW APPROACH[1]

Thomas Y.C. Woo and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

In most existing systems, authorization is specified using some low-level system-specific mechanisms, e.g., protection bits, capabilities and access control lists. We argue that authorization is an independent semantic concept that must be separated from implementation mechanisms and given a precise semantics. We propose a logical approach to representing and evaluating authorization. Specifically, we introduce a language for specifying *policy bases*. A policy base encodes a set of authorization requirements and is given a precise semantics based upon a formal notion of *authorization policy*. The semantics is computable, thus providing a basis for authorization evaluation.

1 Introduction

To guarantee the security of a distributed system, many concerns need to be addressed. These include authentication, authorization, auditing, accounting and availability, among others. In this paper, we propose a new foundation for authorization, specifically, one that is appropriate for the design and implementation of distributed systems.

The problem of authorization can be divided into two related subproblems: *representation* and *evaluation*. Representation refers to the specification of authorization requirements, while evaluation refers to the actual determination of the authorities of *subjects* given the authorization requirements. The authority

[1]This work was supported in part by NSA INFOSEC University Research Program under contract no. MDA 904-92-C5150 and in part by National Science Foundation grant no. NCR-9004464. A preliminary version of this paper was presented in [37]. Published in *Journal of Computer Security*, 1993. Postscript files of this and other papers of the Networking Research Laboratory are available from http://www.cs.utexas.edu/~lam/NRL.

of a subject is its rights to access *objects*. (Thus our view of authorization is limited to access control; we do not consider issues of covert channels and secure information flow [7, 13, 22].)

Conceptually, the rights of subjects to access objects can be stored in an *access matrix* [14, 20, 21], with rows corresponding to subjects, columns corresponding to objects, and matrix entries indicating various *access rights*. (See examples in Section 3.) Practical implementations of an access matrix usually take advantage of the sparseness of the matrix, and are based upon capabilities (access rights stored by row), access control lists (access rights stored by column), or some hybrid combination of these approaches [7, 9].

Distributed systems and the prevalent client-server style of computing give rise to new problems in the specification of authorization requirements. For examples:

- New kinds of attributes need to be considered. For instance, an authorization requirement in a distributed system may include the *location* of a subject as an attribute in addition to the identity of the subject. That is, it is possible that a subject U is authorized to update a file F from node N but not from another node N'. Other attributes include: the role a subject is assuming, the groups a subject belongs to, any delegations a subject may have, and such.
- A large-scale distributed system is typically composed of multiple independent domains, which are managed by possibly different administrative authorities. In fact, even a single domain may have several security administrators. In these situations, authorizations in one domain may affect those in other domains in unexpected ways. For instance, let X, Y and Z be three independent domains within a distributed system administered respectively by authorities A, B and C. Suppose A authorizes requests from Y to access resources in X but denies requests from Z. If B authorizes requests from Z to access resources in Y, such authorization would indirectly contradict the one by A, because a user in Z might be able to access resources in X by "going through" domain Y.

Existing models of authorization have not been designed to address these problems [16, 23, 30]. Furthermore, existing approaches are unsatisfactory in the following respect: authorization requirements can only be specified using some low-level system-specific mechanisms. For example, in Unix, accesses to the file system are specified by protection bits associated with each file, and authorization is determined by how these protection bits are set. Such embedding of authorization requirements into mechanisms presents serious drawbacks. First, authorization requirements are limited to those that can be specified by these low-level mechanisms. Second, the semantics of authorization is dependent on the semantics of the low-level mechanisms, which is not formally defined and indeed may vary from one implementation to another.[2] This poses problems in large-scale distributed systems with heterogeneous implementations.

For example, many people have recognized the limitations of protection bits in Unix and have proposed various ad-hoc extensions to it. Each of these extensions addresses one type of authorization requirements or another without solving the above problems as a whole. Furthermore, there can be subtle interactions among these extensions, which may render a security administrator unable to comprehend what actually has been authorized.[3] In fact, such confusion can be a major source of security violations.

The separation of *policies* from *mechanisms* has long been recognized as a fundamental tenet in system design [18, 33]. A policy specifies what is required, while a mechanism provides the actual enforcement. In the context of authorization, this means that a policy of authorization should have an independent semantics that is separated from its implementation in system-specific mechanisms. To this end, we advocate a language-based approach to authorization. For representation, we need a language that is expressive enough for specifying commonly encountered authorization requirements. The language must be given a formal semantics so that the meaning of an authorization requirement stated using the language can be precisely determined. This way, a security administrator is able to reconcile easily between what he intends to authorize with what he has actually authorized.

With this approach, authorization evaluation reduces to computation of semantics. The complexity of such computation is highly dependent on the particular language used. The computation mechanism can range from a trivial table lookup (e.g., if the language is simply an access matrix) to a full-fledged theorem proving procedure (e.g., if the language is first order logic). In general, the more expressive the representation language, the more complex the computation mechanism. Thus issues of representation and evaluation must be examined hand in hand with careful consideration of various tradeoffs.

In this paper, we propose a new foundation for representing and evaluating authorization. Our contributions are as follows. We first identify three types of structural properties inherent in authorization requirements. We argue that such structural properties can be effectively exploited to reduce the complexity of representing and evaluating authorization in large-scale distributed systems. We introduce a representation language in which the structural properties can be represented in a straightforward manner. The language is designed to specify *policy bases.* A policy base encodes a set of authorization requirements and is given a precise semantics based upon a formal notion of *authorization policy.* The semantics is computable via a translation to *extended logic programs* (see

[2]A vivid example of this is the assortment of *setuid/setgid* function calls available in different flavors of Unix.

[3]See for example [19] and the POSIX Security Draft Standard P1003.6 which discuss how to supplement Unix protection bits with access control lists.

Theorem B in Section 7.3), thus providing a evaluation mechanism based on the interpretation of extended logic programs.

The balance of this paper is organized as follows. In Section 2, we compare our work to related research. In Section 3, we identify three types of structural properties in authorization requirements. In Section 4, we discuss language requirements for representing authorization. In Section 5, we present our model of authorization. In Section 6, we introduce authorization policy as a semantic notion. In Section 7, we introduce our language for specifying policy bases and describe its syntax and semantics as well as some guidelines for its usage. In Section 8, we provide some examples of policy bases, including the *Bell-LaPadula* model [3] and some inheritance rules. In Section 9, we discuss implementation considerations. This section is necessarily brief, and is intended only to give a general idea of how our framework can be put into practice. We are currently building a prototype implementation based on the ideas presented in this paper. The details of our implementation will be reported in a future paper. Lastly, in Section 10, we provide some concluding remarks.

2 Relation to Other Work

Before relating this paper to other work, we would like to emphasize several points. First, our work is concerned with *access control*, and does not address *information flow control* [4, 13, 28]. Thus, the typical concerns in most security modeling work [12, 13, 27, 29] are orthogonal to the ones in this paper. In particular, these references focus on modeling the abstract security properties of a system as a whole, while our work has a more narrow focus on authorization only.

Second, the research reported in this paper is mainly concerned with representation and evaluation issues of *static* authorization requirements, which are to be satisfied in each individual state. In other words, we do not model the dynamics of authorization. In this sense, the model we use and the issues we investigate are very different from those studied in [16, 17, 25, 34, 35, 36]. For example, we do not study the problem of access rights propagation, commonly known as *safety analysis* [5, 6, 17]. Similarly, the creation and deletion of subjects and objects are not modeled within our framework. We stress, however, that this does not mean that our framework cannot be extended to handle these issues. In this paper, we choose not to pursue these extensions because we are interested in other issues.

In relating this paper to previous work, we observe that the work by Lunt [26] is most relevant to us. She raised a similar question of ambiguity in the interpretation of authorization policies. In particular, she examined different interpretations of denial and several conflict resolution schemes. Her focus was more in identifying and understanding the problems. In this paper, we put forth specific constructs that allow an administer to explicitly specify and differentiate these interpretations and schemes.

The paper by Abadi, et al. [1] deals also with access control in a distributed system setting. They also make use of a logic, specifically, a modal logic together with a *calculus of principals*. Their goal, however, is different from ours. Their logic is used to explain the meaning of *roles* and *delegation*, and also the operation of certain protocols. They do not study representation issues. In particular, the concept of a *statement* (standing for a specific access request) in their logic is fully abstract (i.e., uninterpreted) [1, p. 725]. In some sense, our work is complementary to theirs in that we investigate the structure of these statements and provide meanings to them.

Lastly, concrete models such as those proposed in [8, 15, 24] address the same general concerns as ours, but for application-specific domains. Our framework can be used as a general basis underlying their respective specific proposals.

3 Three Types of Structural Properties

Authorization requirements are highly structured because the set of subjects and the set of objects in a system are usually highly structured. For example, users belonging to the same working groups are likely to share similar authorizations; while objects pertaining to a common task are usually given similar authorizations.

To illustrate such structures, we look at some examples. Consider the authorization specified by the following access matrix:

	$P.src$	$P.exe$	$P.doc$
A	r, w	e, w	r, w
B		e	r

Subject A, who is the developer of software P, can read/write the source file $P.src$, execute and write the executable file $P.exe$, and read/write the documentation file $P.doc$; while subject B, who is a user of P, is only allowed to execute $P.exe$ and read $P.doc$. Certain structures in the authorization are readily apparent:

(1) A, being the developer of P, must be able to update all of the files related to P, i.e., A must have write access to all three files. Similarly, B should be allowed to read the documentation $P.doc$ if he is allowed to execute $P.exe$.

(2) Denials of access rights are represented implicitly by their absence in an entry. (Thus explicit denials are not possible and moreover, a denial is indistinguishable from a lack of information about an authorization.)

We call the structures exhibited in (1) *closure* properties among authorizations. In general, a closure property stipulates that a set of authorizations should either be simultaneously authorized or denied, because a partial authorization produces an "unusable" system. Closure properties can be used to ensure the "consistency" of authorizations as in the above example or to derive new authorizations from existing ones.

The structure exhibited in (2) is called a *default* property. A default property can be used as a convention to represent implicit knowledge as in the example above (i.e., absence implies denial) or as a deduction rule when information is incomplete. In fact, most real systems employ default properties in one way or another. For example, two kinds of policies are typically used: a *restrictive* policy is one whereby a request is denied unless explicitly authorized and a *permissive* policy is one whereby a request is always granted unless explicitly denied. Both make use of default properties.

We now turn to another example. Consider the authorization specified by the following access matrix (where, for an access right a, we use $\neg a$ to denote its explicit denial):

	$F.1$	$F.2$	F	H
A			e	
G_1	r	w		r
G_2	r	$\neg w$	$\neg e$	

Suppose A is a member of groups G_1 and G_2, and $F.1$ and $F.2$ are two components of an object F (e.g., two tables in a database). Several questions can be asked about the authorization:

(1) G_1 is authorized to read $F.1$. Is A, who is a member of G_1, also authorized to do the same? This is easy to resolve as both groups to which A belongs are allowed to read $F.1$; hence, A should be authorized to read $F.1$. Consider now file H for which only one of the two groups is authorized to read. Is A authorized to read H? The answer is not obvious. So is $F.2$ for which G_1 and G_2 have been given opposite authorizations.

(2) Consider object F. A is authorized to execute F. However, G_2 to which A belongs is explicitly denied the same access. Does the denial of G_2 revoke the authorization of A? Or does A's explicit authorization override the denial of G_2?

All of the above questions can be answered by precisely defining another kind of structural properties, called *inheritance* properties. Inheritance is especially important in large-scale distributed systems where the granularity of authorization ranges from an individual to an entire domain. Inheritance properties are used to relate authorizations specified with these different granularities.

In sum, it is important that the structural properties described in this section be exploited to obtain succinct representation and efficient evaluation of authorization requirements.

4 Language Requirements

From the above discussions, a language for representing authorization requirements should satisfy the following criteria:

- It should be declarative and have a semantics that is independent of implementation mechanisms.
- The semantics should be efficiently computable, hence allowing efficient authorization evaluation.
- It should allow easy expression of the closure, default and inheritance properties discussed in Section 3.

In the following, we discuss four more requirements for such a language.

First, authorization is *nonmonotonic.* That is, if a set of authorization requirements is augmented by a new requirement, a subject who was previously allowed access to an object may no longer be allowed the same access. A good example of such nonmonotonicity arises in the use of defaults. For example, suppose the set of authorization requirements includes the following default:

if s is not explicitly denied read access to o
then by default s is allowed read access to o

If the set of requirements is later augmented with an explicit requirement denying s read access to o, the previous grant should be retracted. Thus, the semantics of a language for authorization must allow such nonmonotonic behavior.

Second, authorization may be *incomplete.* That is, there may be authorization requests such that insufficient information is available to determine if they should be granted or denied. Such incompleteness should be allowed in the semantics of a language for authorization. There are two reasons:

- An incompleteness may be the result of an oversight or error on the part of security administrators. Thus when an incompleteness is detected, it can serve as an alarm signalling potentially more serious problems. Therefore, it is advantageous that such incompleteness not be masked out automatically by the language semantics.

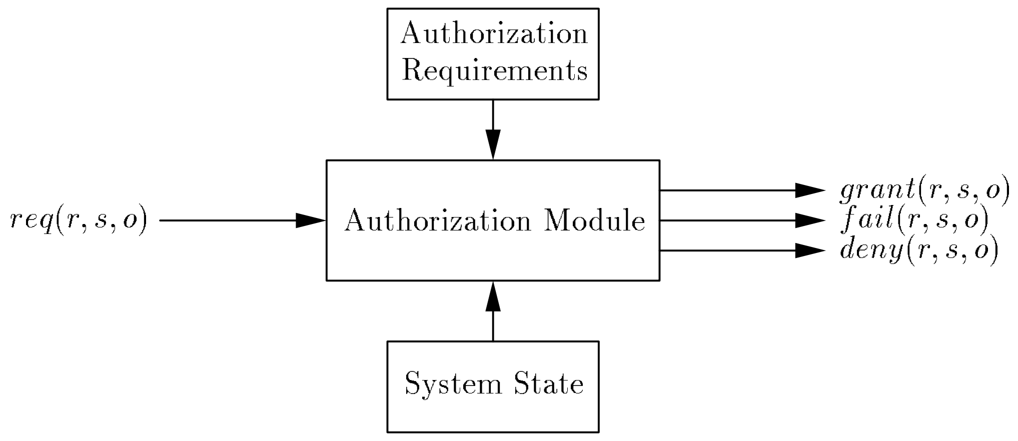

Figure 1: Model of authorization

- An incompleteness may be intentional so that it can be "filled in" later when composition is performed (see below). Thus, it is important that such intentional incompleteness be allowed by the language semantics.

Note that this strictly generalizes the idea of a *reference monitor* [9], where no incompleteness is allowed.

Third, authorization may be *inconsistent*. That is, it is possible for an authorization request to be both granted and denied. The reasoning is similar to that of incomplete authorization: An inconsistent authorization may signal errors on the part of security administrators or they can arise from the composition of authorization requirements, especially in a large scale distributed system. Therefore, the semantics of a language for authorization must be able to handle inconsistencies.

Fourth, multiple authorities may coexist in a distributed system environment. These authorities can be peers who coadminister a system or they can be hierarchically related in a supervisor-subordinate fashion. Each of them may contribute authorization requirements pertinent to the part of the system he is concerned with. The authorization of the entire system is a *composition* of these individually contributed authorization requirements. Thus a language for authorization should include operators for composing authorization requirements.

5 Our Model

Our model of authorization is shown in Figure 1. Before a subject s can perform a particular access r on an object o, s must first obtain the access right r for o.

Subject s does so by submitting a request of the form $req(r, s, o)$ to the *authorization module*, which responds with $grant(r, s, o)$, $deny(r, s, o)$ or $fail(r, s, o)$. A $grant(r, s, o)$ is returned if the authorization module can determine that s is authorized to have r access to o, while a $deny(r, s, o)$ is returned if the authorization module can determine that s is denied r access to o. A $fail(r, s, o)$ is returned if the authorization module fails to establish either one of the previous two cases.

To make an authorization decision, the authorization module consults the authorization requirements and the system state. The system state is needed for authorization requirements that contain system state variables as parameters. Some examples of this kind of authorization requirements are "At most 5 copies of a program P can be running concurrently in all nodes of the system" and "User A is allowed to execute program P only if the current system load is less than 2".

In our model, an authorization requirement is stated as a rule and a collection of such rules constitutes a policy base (see Section 7). The authorization module is an interpreter which takes as input a policy base B, the current system state, and a request $req(r, s, o)$, and tries to verify that either $grant(r, s, o)$ or $deny(r, s, o)$ "follows" from the semantics of B given the current system state. If $grant(r, s, o)$ follows, the request is granted. If $deny(r, s, o)$ follows, the request is denied. If neither follows, a $fail(r, s, o)$ is returned. The pathological case in which both $grant(r, s, o)$ and $deny(r, s, o)$ follow can be resolved by enforcing certain priorities between grants and denials. A precise definition of the "follows" relation is given by a formal semantics of policy bases to be presented below.

Note that Figure 1 is actually a simplified picture of our model. In general, the policy bases can be located in different parts of a distributed system, and multiple instantiations of the authorization module can be running concurrently across the system. More discussions on implementation are provided in Section 9.

6 Authorization Policy

Informally, an authorization policy is the set-theoretic equivalent of an access matrix. Its precise meaning is defined in what follows.

Definition. An *authorization policy* (or *policy* in short) over a set of subjects S, a set of objects O and a set of access rights R is a 4-tuple (P^+, P^-, N^+, N^-) where each component is a subset of $\{(r, s, o) \mid r \in R, s \in S, o \in O\}$. □

The intuitive meaning of a policy $A = (P^+, P^-, N^+, N^-)$ is as follows: P^+ records the rights that are explicitly granted, i.e., if $(r, s, o) \in P^+$, subject s is explicitly granted access right r to object o. Similarly, N^+ records the rights that are explicitly denied. P^- (N^- respectively) records those rights that should not be explicitly granted (denied respectively) under this policy. P^- and N^- are useful for defining the semantics of policy composition.

A policy $A = (P^+, P^-, N^+, N^-)$ is *sound* if there does not exist a triple $t = (r, s, o)$ such that $t \in P^+ \cap P^-$ or $t \in N^+ \cap N^-$. A policy $A = (P^+, P^-, N^+, N^-)$ is *strongly sound* if it is sound and $P^+ \cap N^+ = \emptyset$.

A policy $A = (P^+, P^-, N^+, N^-)$ is *complete* if for all $s \in S$, $o \in O$ and $r \in R$, $(r, s, o) \in P^+ \cup P^- \cup N^+ \cup N^-$. A policy $A = (P^+, P^-, N^+, N^-)$ is *strongly complete* if it is complete and both P^- and N^- are empty. Thus it is sufficient to represent a strongly complete policy with an ordered pair (P, N).

Given a strongly sound policy $A = (P^+, P^-, N^+, N^-)$, we can define three *authorization relations* between A and a triple (r, s, o):

$$\begin{aligned} A \textit{ grants } (r,s,o) &\quad \text{iff} \quad (r,s,o) \in P^+ \\ A \textit{ denies } (r,s,o) &\quad \text{iff} \quad (r,s,o) \in N^+ \\ A \textit{ fails } (r,s,o) &\quad \text{iff} \quad (r,s,o) \notin P^+ \cup N^+ \end{aligned}$$

Authorization evaluation can proceed as follows: Given a request from a subject s for access r to an object o, $grant(r, s, o)$ is returned if A *grants* (r, s, o), $deny(r, s, o)$ is returned if A *denies* (r, s, o) and $fail(r, s, o)$ is returned if A *fails* (r, s, o). Note that if A is also strongly complete, then $fail(r, s, o)$ would never be returned.

7 Policy Base

In this section, we present a language for stating authorization requirements in policy bases. The language is essentially a many-sorted first-order language with a *rule* construct. The rule construct is similar to the default construct in *default logic* [32]; however, we give it a different semantics. The rule construct is useful for stating structural properties of authorization requirements.

From some domain-specific considerations, we impose several restrictions on the kind of first-order formulas allowed. We briefly describe the restrictions and their motivations:

- We desire to have a computable semantics. As validity in an infinitary theory is typically semi-decidable, we restrict ourselves to finitary theories. To achieve this, we do not allow function symbols in our language and postulate only finite sets of access-right, subject and object constants. This also allows us to eliminate quantifications.

  We note that this finiteness assumption only requires that at any particular time, the sets of subjects, objects and access rights are finite. It does not imply that the sets are fixed. In particular, subjects and objects can be dynamically created and deleted. However, we have chosen not to model such creation and deletion in our present framework. Instead, our focus is on static authorization requirements.

Furthermore, the finiteness assumption allows us to effectively reduce *open* policy bases into *closed* ones (see below). This is analogous to the so called *domain closure* or *closed world* assumption typically used in database research.

- We allow the use of disjunction only in highly restricted ways. For example, we cannot state in our language the authorization requirement "Subject A is either allowed to read file F or write file G". Neither can we state "There is a subject x who can read file F" in our language. Our view is that such disjunctive authorization requirements provide insufficient information for determining the exact extent of authorization.

  On a closer look, this limitation is not as restrictive as it seems. In a realistic authorization policy, disjunctive authorization requirements are stated mostly for convenience and their disjunctive nature is usually immediately resolved when other requirements are taken into consideration. This is analogous to the case in classical logic where the statement $A \vee B$ when combined with $\neg A$ yields B, which is non-disjunctive. Purely disjunctive authorization requirements are rare and counterintuitive.

7.1 Syntax

The alphabet of our language is derived from the system to be modeled. Consider a system with S as its set of subjects, O its set of objects and R its set of access rights. (Note that S, O and R are all finite sets.) We postulate the following alphabet for our language:

- a set of *ordinary* variables $\mathcal{V}$,
- a set of *propositional* variables $\mathcal{P}$,
- two propositional constants T and F,
- a finite set of subject constants $\mathcal{S}$,
- a finite set of object constants $\mathcal{O}$,
- a finite set of binary predicate symbols $\mathcal{R} = \{r^+, r^- \mid r \in R\}$,
- two special predicate symbols "=", and "$\in$".

The set $\mathcal{S}$ ($\mathcal{O}$) contains a constant symbol for each subject (object) in S (O). In other words, each subject or object in the system is explicitly represented by a constant symbol in the language.

A *term* is an ordinary variable, a subject constant or an object constant. An *atom* is a propositional constant, a propositional variable or a predicate $p(t, t')$ where p is a predicate symbol and t, t' are terms. We adopt the convention of writing predicates involving $=$ or $\in$ in the infix form, i.e., we write $=(t, t')$ as $t = t'$ and $\in(t, t')$ as $t \in t'$. An atom formed from a predicate symbol in $\mathcal{R}$ is called a *distinguished* atom; and the rest *ordinary* atoms.

A *literal* is an atom or the negation of an atom. Negation is denoted by the symbol $\neg$. A literal formed from a distinguished atom is called a *distinguished* literal, while a literal formed from an ordinary atom is called an *ordinary* literal. A literal is *positive* if it is an atom, and *negative* if it is the negation of an atom. Let a be an atom, then the two literals a and $\neg a$ are called *complementary* literals. We define $\overline{a}$ to be $\neg a$ and $\overline{\neg a}$ to be a. Thus, ℓ and $\overline{\ell}$ are always complementary for any literal ℓ.

A *formula* is a literal, a conjunction of two formulas f and f', denoted by $f \wedge f'$, or a disjunction of two formulas f and f', denoted by $f \vee f'$. A *basic formula* is a formula that only contains propositional constants and distinguished literals. A subclass of basic formulas that does not contain disjunctions is called *conjunctive formulas.* Note that in our formulas, unlike those of first order logic, negation occurs only at the level of literals. A formula is *closed* if it does not contain ordinary variables, otherwise it is *open.*

A *rule* is written in the form $\frac{f\,:\,f'}{g}$ where f is a formula, f' a basic formula and g a conjunctive formula. f, f' and g are respectively called the *prerequisite*, *assumption* and *consequent* of the rule.

Notation. To simplify our presentation, we introduce a syntactic operator *neg* for basic formulas. The definition of *neg* is as follows:

- $neg(\mathsf{T})$ is F,
- $neg(\mathsf{F})$ is T,
- $neg(f)$ is $\overline{f}$, if f is a literal,
- $neg(f_1 \wedge f_2)$ is $neg(f_1) \vee neg(f_2)$,
- $neg(f_1 \vee f_2)$ is $neg(f_1) \wedge neg(f_2)$.

Thus the effect of the *neg* operator is similar to that of applying negation to the entire basic formula and then pushing it inward using De Morgan's law. □

Convention. To be succinct, we use several abbreviations. First, if any component formula is missing from a rule, it is assumed to be T. Second, we use the notation $f \Rightarrow g$ to represent a rule of the form $\frac{f\,:\,\mathsf{T}}{g}$. Third, $\mathsf{T} \Rightarrow g$ is further abbreviated to g. □

Example 1. Let $\mathcal{V} = \{x, y, \ldots\}$, $\mathcal{P} = \{p, q\}$, $\mathcal{S} = \{\mathsf{A}, \mathsf{B}, \mathsf{G}\}$, $\mathcal{O} = \{\mathsf{X}, \mathsf{Y}, \mathsf{Z}\}$, and $R = \{\mathsf{read}, \mathsf{write}\}$. Then the following are rules:

$$\mathsf{read}^-(\mathsf{G}, x)$$

$$\mathsf{read}^+(\mathsf{A}, \mathsf{X}) \Rightarrow \mathsf{read}^+(\mathsf{A}, \mathsf{Y})$$

$$x \in \mathsf{G} \wedge \mathsf{read}^-(\mathsf{G}, \mathsf{Y}) \Rightarrow \mathsf{read}^-(x, \mathsf{Y})$$

$$\neg p \vee \mathsf{write}^+(x, \mathsf{Z}) \Rightarrow \neg\mathsf{read}^+(x, y)$$

$$\frac{p \wedge \mathsf{read}^+(x, \mathsf{Z}) : \mathsf{read}^+(x, \mathsf{Y})}{\mathsf{read}^+(x, \mathsf{Y})}$$

$$\frac{x \in G \wedge \mathsf{write}^+(\mathsf{G}, y) : \mathsf{write}^+(x, y) \wedge \neg\mathsf{write}^-(x, y)}{\mathsf{write}^+(x, y)}$$

□

Definition. A *policy base* (or *base* in short) is a finite set of rules. □

A rule $\frac{f\,:\,f'}{g}$ is *closed* if f, f' and g are all closed; otherwise it is *open*. A rule is *pure* if f is also a basic formula. A base is *closed* if it contains only closed rules. A base is *pure* if it contains only pure rules.

7.2 Semantics for Closed Policy Base

We present a semantics for closed bases here. The semantics for open bases is in Section 7.4. Every mention of base in this subsection is taken to mean a closed base unless explicitly stated otherwise.

The semantics of a base is given by its *extensions*. An extension is a set of distinguished literals[4]. An extension provides a straightforward interpretation for distinguished literals: such a literal is true if and only if it is contained in the extension. An extension is similar in concept to a model in the standard semantics for classical logic.

An extension naturally defines a policy. More precisely, let Σ be an extension and let

$$\begin{aligned} P^+ &= \{(r, s, o) \mid r^+(s, o) \in \Sigma\} \\ P^- &= \{(r, s, o) \mid \neg r^+(s, o) \in \Sigma\} \\ N^+ &= \{(r, s, o) \mid r^-(s, o) \in \Sigma\} \\ N^- &= \{(r, s, o) \mid \neg r^-(s, o) \in \Sigma\} \end{aligned}$$

[4] An extension is similar to a Herbrand base except that it contains literals instead of just atoms.

Clearly, (P^+, P^-, N^+, N^-) is a policy. We call it the *policy defined by* Σ, and denote it by $\alpha(\Sigma)$. This establishes a one-one correspondence between an extension and a policy.

To provide meanings for ordinary literals, we use an *assignment* function $\mathcal{I} : \mathcal{P} \mapsto \{\mathsf{true}, \mathsf{false}\}$ and a *group relation* $\mathcal{G}$. An assignment function provides interpretation for propositional variables, while a group relation provides interpretation for the predicate "$\in$"; they together model the system state. The equality predicate "$=$" is interpreted as the identity relation. We also adopt the *unique names assumption*, i.e., $c \neq c'$ for all c, c' in $\mathcal{S} \cup \mathcal{O}$.

Before we give our definition for extension, we first define a *satisfaction* relation between a set Σ of distinguished literals and a closed formula f with respect to an assignment $\mathcal{I}$ and a group relation $\mathcal{G}$. We denote the satisfaction relation by $\Sigma \models_{\mathcal{I},\mathcal{G}} f$. The definition is by structural induction:

- f is a propositional constant, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff f is T
- f is a propositional variable, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $\mathcal{I}(f) = \mathsf{true}$
- f is $t = t'$, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $t \equiv t'$
- f is $t \in t'$, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $(t, t') \in \mathcal{G}$
- f is a distinguished literal L, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $L \in \Sigma$
- f is $\neg f'$ where f' is an ordinary atom, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $\Sigma \not\models_{\mathcal{I},\mathcal{G}} f'$
- f is $f_1 \wedge f_2$, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $\Sigma \models_{\mathcal{I},\mathcal{G}} f_1$ and $\Sigma \models_{\mathcal{I},\mathcal{G}} f_2$
- f is $f_1 \vee f_2$, then
  $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $\Sigma \models_{\mathcal{I},\mathcal{G}} f_1$ or $\Sigma \models_{\mathcal{I},\mathcal{G}} f_2$

It should be clear that for a basic formula f, the satisfaction relation is independent of $\mathcal{I}$ and $\mathcal{G}$. More precisely, let f be a basic formula, $\mathcal{I}, \mathcal{I}'$ assignments, $\mathcal{G}, \mathcal{G}'$ group relations and Σ a set of distinguished literals. Then $\Sigma \models_{\mathcal{I},\mathcal{G}} f$ iff $\Sigma \models_{\mathcal{I}',\mathcal{G}'} f$. We would abbreviate $\models_{\mathcal{I},\mathcal{G}}$ to $\models$ in this case.

Note that our semantics is different from the standard semantics for classical logic in several ways. First, $F \wedge \neg F$ represents a contradiction in classical logic and hence does not admit any model. In our case, we have $\{F, \neg F\} \models_{\mathcal{I},\mathcal{G}} F \wedge \neg F$. Second, in classical logic, if Σ satisfies both $F \vee G$ and $\neg F$, then it must also satisfy G. This is not true in our semantics as $\{F, \neg F\} \models_{\mathcal{I},\mathcal{G}} F \vee G$ and $\{F, \neg F\} \models_{\mathcal{I},\mathcal{G}} \neg F$, but $\{F, \neg F\} \not\models_{\mathcal{I},\mathcal{G}} G$. A semantics that exhibits such non-classical behavior is often called *paraconsistent.*

Given a base, we are now ready to define its extensions. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ a group relation. We define an operator $\Gamma_{B,\mathcal{I},\mathcal{G}}$ that given a set of distinguished literals, returns a new set of distinguished literals. The formal definition of $\Gamma_{B,\mathcal{I},\mathcal{G}}$ is as follows: Let Σ be a set of distinguished literals. Define

$$S^{\mathcal{I},\mathcal{G}}_{B,\Sigma} = \left\{ M \;\middle|\; \begin{array}{l} M \text{ is a set of distinguished literals and} \\ \text{for all } \frac{f\,:\,f'}{g} \in B, \text{ if } M \models_{\mathcal{I},\mathcal{G}} f \text{ and } \Sigma \not\models neg(f') \text{ then } M \models g \end{array} \right\}$$

then

$$\Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma) = \text{ the intersection of all elements in } S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$$

The intuitive meaning of a rule $\frac{f\,:\,f'}{g}$ is as follows: If a set Σ of distinguished literals satisfies f, and there is no evidence that the negation of f' is satisfied (hence it is consistent to assume that Σ satisfies f'), then Σ must also satisfy g. Note that both $neg(f')$ and g are basic formulas. Hence we write $\models$ instead of $\models_{\mathcal{I},\mathcal{G}}$ in the condition for $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$.

In the case that B is a pure base, it should be clear from the above definitions that $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$ and hence the operator $\Gamma_{B,\mathcal{I},\mathcal{G}}$ are independent of $\mathcal{I}$ and $\mathcal{G}$. That is, $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma} = S^{\mathcal{I}',\mathcal{G}'}_{B,\Sigma}$ $\Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma) = \Gamma_{B,\mathcal{I}',\mathcal{G}'}(\Sigma)$ for any set of distinguished of literals Σ. In the following, we would denote them respectively by $S_{B,\Sigma}$ and Γ_B.

We also make the following observation: Let

$$CON(B) = \left\{ g \;\middle|\; \frac{f\;:\;f'}{g} \in B \right\}$$

Then each element of $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$ is a subset of $CON(B)$. Hence $\Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma) \subseteq CON(B)$. Note that $CON(B)$ is finite, as B is finite.

Definition. Let B be a base, $\mathcal{I}$ an assignment, $\mathcal{G}$ a group relation and Σ a set of distinguished literals. Σ is an *extension* of B *under* $\mathcal{I}$ and $\mathcal{G}$ if $\Sigma = \Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma)$, i.e., Σ is a fixed point of the operator $\Gamma_{B,\mathcal{I},\mathcal{G}}$. □

In the case of a pure base, the fixpoints are independent of the assignment and the group relation. Therefore, we can just refer to them as extensions of B.

From the above observation that $\Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma)$ is a subset of $CON(B)$ and that $CON(B)$ is finite, a simple procedure for finding all extensions of a base B is to enumerate all subsets of $CON(B)$ and check each one for satisfaction of the fixed point equation.

Since each extension of a base defines a policy, in the case that B admits a *unique* extension Σ under $\mathcal{I}$ and $\mathcal{G}$, the policy defined by Σ can be taken to be the semantics of B under $\mathcal{I}$ and $\mathcal{G}$. We formalize this in the following definition.

Definition. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ group relation. Suppose B admits a unique extension Σ under $\mathcal{I}$ and $\mathcal{G}$. Then $\alpha(\Sigma)$, the *policy determined by B under* $\mathcal{I}$ and $\mathcal{G}$, will be denoted by $\mathcal{E}_{\mathcal{I},\mathcal{G}}(B)$. □

The authorization relations introduced in Section 6 can be naturally extended to a base as follows. (Note that this is well defined only when $\mathcal{E}_{\mathcal{I},\mathcal{G}}(B)$ itself is well-defined and is a strongly sound policy.)

Definition. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ group relation. Let $s \in S$, $o \in O$ and $r \in R$:

$$\begin{aligned} B \textit{ grants } (r,s,o) \textit{ under } \mathcal{I},\mathcal{G} &\quad \text{iff} \quad \mathcal{E}_{\mathcal{I},\mathcal{G}}(B) \textit{ grants } (r,s,o) \\ B \textit{ denies } (r,s,o) \textit{ under } \mathcal{I},\mathcal{G} &\quad \text{iff} \quad \mathcal{E}_{\mathcal{I},\mathcal{G}}(B) \textit{ denies } (r,s,o) \\ B \textit{ fails } (r,s,o) \textit{ under } \mathcal{I},\mathcal{G} &\quad \text{iff} \quad \mathcal{E}_{\mathcal{I},\mathcal{G}}(B) \textit{ fails } (r,s,o) \end{aligned}$$

These three relations represent the authorization defined by a base B, and are taken to be its semantics. □

Clearly, the above semantics is well-defined only when B admits a unique extension under $\mathcal{I}$ and $\mathcal{G}$. However, as shown in the examples below, this unique extension property is not true in general.

Example 2. Consider the base

$$B_1 = \left\{ \frac{:\ \mathsf{read}^+(\mathsf{A},\mathsf{X}) \wedge \neg\mathsf{read}^+(\mathsf{A},\mathsf{Y})}{\mathsf{read}^+(\mathsf{A},\mathsf{X})}, \frac{:\ \mathsf{read}^+(\mathsf{A},\mathsf{Y}) \wedge \neg\mathsf{read}^+(\mathsf{A},\mathsf{Z})}{\mathsf{read}^+(\mathsf{A},\mathsf{Y})}, \frac{:\ \mathsf{read}^+(\mathsf{A},\mathsf{Z}) \wedge \neg\mathsf{read}^+(\mathsf{A},\mathsf{X})}{\mathsf{read}^+(\mathsf{A},\mathsf{Z})} \right\}$$

B_1 does not admit any extension under all assignments and group relations. This can be shown as follows. First,

$$CON(B_1) = \left\{\mathsf{read}^+(\mathsf{A},\mathsf{X}), \mathsf{read}^+(\mathsf{A},\mathsf{Y}), \mathsf{read}^+(\mathsf{A},\mathsf{Z})\right\}$$

Any extension of B_1 must be a subset of $CON(B_1)$. We check each such subset to see if it satisfies the fixed point equation. We group these subsets into four cases: (1) The empty set. We have $\Gamma_{B_1}(\emptyset) = CON(B_1) \neq \emptyset$. (2) The singleton sets: $\Sigma_1 = \{\mathsf{read}^+(\mathsf{A},\mathsf{X})\}$, $\Sigma_2 = \{\mathsf{read}^+(\mathsf{A},\mathsf{Y})\}$ or $\Sigma_3 = \{\mathsf{read}^+(\mathsf{A},\mathsf{Z})\}$. For Σ_1, we have $\Gamma_{B_1}(\Sigma_1) = \Sigma_1 \cup \Sigma_2 \neq \Sigma_1$. The calculations for Σ_2 and Σ_3 are similar. (3) The two-element sets: $\Sigma_4 = \{\mathsf{read}^+(\mathsf{A},\mathsf{X}), \mathsf{read}^+(\mathsf{A},\mathsf{Y})\}$, $\Sigma_5 = \{\mathsf{read}^+(\mathsf{A},\mathsf{Y}), \mathsf{read}^+(\mathsf{A},\mathsf{Z})\}$ or $\Sigma_6 = \{\mathsf{read}^+(\mathsf{A},\mathsf{Z}), \mathsf{read}^+(\mathsf{A},\mathsf{X})\}$. For Σ_4, we have $\Gamma_{B_1}(\Sigma_4) = \Sigma_2 \neq \Sigma_4$. The calculations for Σ_5 and Σ_6 are similar. (4) The set $CON(B_1)$. We have $\Gamma_{B_1}(CON(B_1)) = \emptyset \neq CON(B_1)$.

Since B_1 contains no ordinary literal, the assignment or group relation cannot be the cause of its lack of an extension. □

Example 3. Consider the base

$$B_2 = \left\{ \frac{:\ \neg\mathsf{write}^+(\mathsf{A},\mathsf{X})}{\mathsf{write}^+(\mathsf{A},\mathsf{Y})}, \frac{:\ \neg\mathsf{write}^+(\mathsf{A},\mathsf{Y})}{\mathsf{write}^+(\mathsf{A},\mathsf{X})} \right\}$$

B_2 admits two extensions $\{\mathsf{write}^+(\mathsf{A},\mathsf{Y})\}$ and $\{\mathsf{write}^+(\mathsf{A},\mathsf{X})\}$ under all assignments and group relations. □

Example 4. Consider the base

$$B_3 = \left\{ \mathsf{read}^+(\mathsf{A},\mathsf{X}), \frac{p \wedge \mathsf{read}^+(\mathsf{A},\mathsf{X})\ :\ \neg\mathsf{write}^-(\mathsf{A},\mathsf{Z})}{\mathsf{write}^-(\mathsf{A},\mathsf{Z})} \right\}$$

If $\mathcal{I}(p) = \mathsf{false}$ then $\{\mathsf{read}^+(\mathsf{A},\mathsf{X})\}$ is an extension. However if $\mathcal{I}(p) = \mathsf{true}$, B_3 does not admit any extension. This can be easily explained by the fact that the assumption and the consequent of the second rule are inconsistent. Thus $\mathcal{I}$ and $\mathcal{G}$ do affect the extensions (if any) of a base. □

Although these examples demonstrate that the unique extension property is not true in general, they also serve to illustrate a common underlying cause for failure. In the above examples, there is a kind of circularity in the rules involving atoms that occur both positively and negatively. For instance, in Example 3, each of the atoms $\mathsf{write}^+(\mathsf{A},\mathsf{X})$ and $\mathsf{write}^+(\mathsf{A},\mathsf{Y})$ occurs positively in the consequent of one rule but negatively in the assumption of the other rule. The application of one rule would necessarily disable the application of the other rule, thus resulting in two different extensions. However, if priorities are enforced between the two rules (e.g., the derivation of $\mathsf{write}^+(\mathsf{A},\mathsf{X})$ is more "important" than the derivation of $\mathsf{write}^+(\mathsf{A},\mathsf{Y})$), then only $\{\mathsf{write}^+(\mathsf{A},\mathsf{X})\}$ would be considered an extension of B_2. This idea can indeed be generalized and a notion of *stratification* can be defined on the set of distinguished atoms, such that a stratified base always possesses a unique extension. We omit the details here and refer the readers to [2, 10, 31].

The semantics of a base can also be given by first "factoring out" the effects of assignments and group relations. We formalize this below.

Let f be a formula, $\mathcal{I}$ an assignment and $\mathcal{G}$ a group relation. Suppose we apply the following transformation to f:

- replace all occurrences of p in f by T if $\mathcal{I}(p) = \mathsf{true}$ and F otherwise
- replace all occurrences of $t \in t'$ in f by T if $(t, t') \in \mathcal{G}$ and F otherwise
- replace all occurrences of $t = t'$ in f by T if $t \equiv t'$ and F otherwise

We denote the resulting formula by $[\mathcal{I}, \mathcal{G}](f)$. It should be clear that if f is a basic formula, $[\mathcal{I}, \mathcal{G}](f) \equiv f$.

Next, we extend the transformation to a rule. Let $d = \frac{f\,:\,f'}{g}$ be a rule. We define:

$$[\mathcal{I}, \mathcal{G}](d) = \frac{[\mathcal{I}, \mathcal{G}](f)\ :\ [\mathcal{I}, \mathcal{G}](f')}{[\mathcal{I}, \mathcal{G}](g)} \equiv \frac{[\mathcal{I}, \mathcal{G}](f)\ :\ f'}{g}$$

The last equality is due to the fact that f' and g are basic formulas. Note that $[\mathcal{I}, \mathcal{G}](d)$ is pure. Lastly, we define

$$[\mathcal{I}, \mathcal{G}](B) = \{[\mathcal{I}, \mathcal{G}](d) \mid d \in B\}$$

Clearly, $[\mathcal{I}, \mathcal{G}](B)$ is a pure base.

Theorem A. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ a group relation. Let Σ be a set of distinguished literals. Then

Σ is an extension of B under $\mathcal{I}$ and $\mathcal{G}$ iff Σ is an extension of $[\mathcal{I}, \mathcal{G}](B)$

Proof. See Appendix A. □

7.3 Computation of $\mathcal{E}_{\mathcal{I},\mathcal{G}}(B)$

For our semantics, authorization evaluation reduces to the computation of $\mathcal{E}_{\mathcal{I},\mathcal{G}}(B)$. In this subsection, we present a semantics-preserving translation of a base B into an *extended logic program* πB, thus reducing the computation of $\mathcal{E}_{\mathcal{I},\mathcal{G}}(B)$ to the computation of πB [11].

We first introduce the concept of an extended logic program. An *extended program clause* is a statement of the form:

$$L \leftarrow L_1, \ldots, L_n, \mathsf{not}\ L_{n+1}, \ldots, \mathsf{not}\ L_m$$

where L and the L_i's are literals. An *extended logic program* is a finite collection of extended program clauses. Extended logic programs are a strict superset of general logic programs, because literals rather than just atoms are allowed in the program clauses.

For extended logic programs, we have developed a paraconsistent semantics (expressed in terms of *models*) using ideas from stable model construction [10]. Our semantics is an extension to the one proposed in [11], and is similarly computable via reduction to general logic programs. A review of this semantics is given in Appendix B.

The essence of our approach is to translate a base into an extended logic program as follows: Let B be a base and let $d \equiv \frac{f : f'}{g}$ be a rule in B. We translate d into the extended program clause

$$g \leftarrow f \wedge not(neg(f'))$$

where *not* is an operator with a definition similar to *neg*:

- $not(\mathsf{T})$ is F,
- $not(\mathsf{F})$ is T,
- $not(h)$ is $\mathsf{not}\ h$, if h is a literal,
- $not(h_1 \wedge h_2)$ is $not(h_1) \vee not(h_2)$,
- $not(h_1 \vee h_2)$ is $not(h_1) \wedge not(h_2)$.

We denote by πB the extended program obtained by applying the above translation to each rule in B.

Theorem B. Let B be a pure base and Σ a set of distinguished literals. Then

$$\Sigma \text{ is an extension of } B \text{ iff } \Sigma \text{ is a model of } \pi B$$

Proof. See Appendix C. □

Corollary C. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ a group relation. Let Σ be a set of distinguished literals. Then

$$\Sigma \text{ is an extension of } B \text{ under } \mathcal{I} \text{ and } \mathcal{G} \text{ iff } \Sigma \text{ is a model of } \pi[\mathcal{I}, \mathcal{G}](B)$$

Proof. Immediate from Theorem A and Theorem B. □

7.4 Semantics for Open Policy Base

Let B be an open base. We view each open rule in B as standing for all its ground instances. In other word, let $d(\bar{x})$ be a rule whose free variables are $\bar{x}$. $d(\bar{x})$ should actually be understood as representing the set of closed rules

$$\{d(\bar{c}) \mid \bar{c} \text{ is a ground substitution for } \bar{x}\}$$

For example, if $\mathcal{S} = \{\mathsf{A}, \mathsf{B}\}$ and $\mathcal{O} = \{\mathsf{X}, \mathsf{Y}\}$, then the open rule

$$\frac{\mathsf{write}^+(x, \mathsf{X}) \;:\; \mathsf{read}^+(x, y)}{\mathsf{read}^+(x, y)}$$

stands for the following set of closed rules:

$$\left\{ \frac{\mathsf{write}^+(\mathsf{A}, \mathsf{X}) \;:\; \mathsf{read}^+(\mathsf{A}, \mathsf{X})}{\mathsf{read}^+(\mathsf{A}, \mathsf{X})}, \frac{\mathsf{write}^+(\mathsf{A}, \mathsf{X}) \;:\; \mathsf{read}^+(\mathsf{A}, \mathsf{Y})}{\mathsf{read}^+(\mathsf{A}, \mathsf{Y})}, \right.$$
$$\left. \frac{\mathsf{write}^+(\mathsf{B}, \mathsf{X}) \;:\; \mathsf{read}^+(\mathsf{B}, \mathsf{X})}{\mathsf{read}^+(\mathsf{B}, \mathsf{X})}, \frac{\mathsf{write}^+(\mathsf{B}, \mathsf{X}) \;:\; \mathsf{read}^+(\mathsf{B}, \mathsf{Y})}{\mathsf{read}^+(\mathsf{B}, \mathsf{Y})} \right\}$$

Thus each open base B can be associated with an "equivalent" closed base B'. The semantics of B is defined to be the same as that for B'.

7.5 Application Guidelines

Having defined the syntax and semantics of policy bases, we now turn to the practical aspects of specifying policy bases. In particular, we provide some guidelines for representing the three kinds of structural properties discussed in Section 3.

Consider a rule $\frac{f \,:\, f'}{g}$. Its intuitive meaning is that the authorization specified by g is allowed if the authorization specified by f is allowed and no authorization contradicting f' has been specified. Informally, f specifies some prerequisite authorization required for g, while f' specifies assumptions that can be used to deduce the authorization specified by g. In the following, we discuss different forms of our rule and show how they can be used.

Consider a rule of the form $\frac{\mathsf{T} \,:\, \mathsf{T}}{g}$, or simply g. Such a rule expresses basic authorization requirements that must be satisfied in a system. There is no prerequisite nor assumption. For example, to say that a user A must be able to read and write his home directory, we write:

$$\mathsf{read}^+(\mathsf{A}, \mathsf{A.home}) \wedge \mathsf{write}^+(\mathsf{A}, \mathsf{A.home})$$

where A.home denotes user A's home directory. These basic authorization requirements form the core upon which other authorizations can be deduced.

A rule of the form $\frac{f\,:\,\mathsf{T}}{g}$, or simply $f \Rightarrow g$, can be used for two purposes. First, it can be used to express a closure property between authorization requirements. For example, consider the rule:

$$\mathsf{execute}^{+}(x, \mathsf{P.exe}) \;\Rightarrow\; \mathsf{read}^{+}(x, \mathsf{P.doc})$$

which says that a user who is authorized to execute a program P.exe should also be allowed to read its associated documentation P.doc.

Another use of the above rule is to define new authorization requirements in terms of others. For example, in Unix, the right to delete a file is equivalent to the right to write the directory containing the file. This can be made explicit as:

$$\mathsf{write}^{+}(x, d) \;\wedge\; f \in d \;\Rightarrow\; \mathsf{delete}^{+}(x, f)$$

where f and d are variables standing for a file and a directory respectively.

Rules can also be used to represent implicit authorizations. There are several reasons why an authorization is left implicit. First, it can be a convention. For example, in general, the number of negative authorizations in a system far exceeds the number of positive ones. Thus for efficiency, a security administrator may specify only the positive ones and leave the negative ones implicit. In other words, the convention is that if a right has not been explicitly authorized, then it is denied. This convention can be formalized in a policy base with the following schema:

$$\frac{:\; r^{-}(s, o)}{r^{-}(s, o)}$$

where $r \in R$.

An inheritance property is another example of implicit authorizations that can be formalized as rules. An example is given in Section 8.

7.6 Specifying Exceptions

In the following, we explore several strategies to specify exceptions. We first introduce the concept of *virtual* rights. Virtual rights are not access rights per se, but are introduced for stating exceptions. We explain this with an example. Suppose we have the following authorization requirements:

> *(1) User* A *is not allowed to write file* X. *(2) A user who is not allowed to write file* X *is also not allowed to read* X *except for those who belong to group G and those who can read file* Y.

As a first attempt, we can express this as two rules:

$$\mathsf{write}^{-}(\mathsf{A}, \mathsf{X})$$
$$\mathsf{write}^{-}(x, \mathsf{X}) \;\wedge\; \neg(x \in \mathsf{G}) \;\wedge\; \neg\mathsf{read}^{+}(x, \mathsf{Y}) \;\Rightarrow\; \mathsf{read}^{-}(x, \mathsf{X})$$

Clearly, these rules correctly represent the requirements. However, they are inflexible and error prone in the following sense: They require every exception to be known and be included in the left hand side of the second rule. Thus for a subject whose exception status is unknown (e.g., subject A above), it will not be explicitly denied the right to read X.

A better way to represent this would be to introduce a virtual right except to represent exceptions and a rule to limit exceptions to the ones explicitly specified.

$$B_4 = \left\{ \begin{array}{l} \mathsf{write}^-(\mathsf{A},\mathsf{X}) \\ \mathsf{write}^-(x,\mathsf{X}) \wedge \neg\mathsf{except}^+(x,\mathsf{X}) \Rightarrow \mathsf{read}^-(x,\mathsf{X}) \\ x \in \mathsf{G} \Rightarrow \mathsf{except}^+(x,\mathsf{X}) \\ \mathsf{read}^+(x,\mathsf{Y}) \Rightarrow \mathsf{except}^+(x,\mathsf{X}) \\ \dfrac{: \neg\mathsf{except}^+(x,\mathsf{X})}{\neg\mathsf{except}^+(x,\mathsf{X})} \end{array} \right\}$$

Subject A is denied read access to X by the second rule in B_4. This is the case because A is assumed to be not an exception by the last rule in B_4. Thus this specification errs on the safe side from a security viewpoint.

Another way of stating the same requirements without using the virtual right except is the following.

$$B_5 = \left\{ \begin{array}{l} \mathsf{write}^-(\mathsf{A},\mathsf{X}) \\ x \in \mathsf{G} \Rightarrow \neg\mathsf{read}^-(x,\mathsf{X}) \\ \mathsf{read}^+(x,\mathsf{Y}) \Rightarrow \neg\mathsf{read}^-(x,\mathsf{X}) \\ \dfrac{\mathsf{write}^-(x,\mathsf{X}) : \mathsf{read}^-(x,\mathsf{X})}{\mathsf{read}^-(x,\mathsf{X})} \end{array} \right\}$$

The main difference between B_4 and B_5 is that in B_4, we only have sufficient conditions for exceptions while in B_5, we can conclude that $\neg\mathsf{read}^-(x,\mathsf{X})$ holds for all excepted individuals.

Yet another way to specify the requirements, one that can be viewed as a hybrid of B_4 and B_5, is the following.

$$B_6 = \left\{ \begin{array}{l} \mathsf{write}^-(\mathsf{A},\mathsf{X}) \\ x \in \mathsf{G} \Rightarrow \mathsf{except}^+(x,\mathsf{X}) \\ \mathsf{read}^+(x,\mathsf{Y}) \Rightarrow \mathsf{except}^+(x,\mathsf{X}) \\ \dfrac{\mathsf{write}^-(x,\mathsf{X}) : \neg\mathsf{except}^+(x,\mathsf{X})}{\mathsf{read}^-(x,\mathsf{X})} \end{array} \right\}$$

B_6 is similar to B_4 in that only sufficient conditions for exceptions are specified. However, for any subject, such as A, who is not explicitly specified as an exception, its exception status remains unknown.

Although B_4, B_5 and B_6 are different with respect to what can be concluded about the excepted individuals, they all have the same semantics with respect to write^- and read^-. In particular, $\mathsf{read}^-(\mathsf{A},\mathsf{X})$ holds in each case.

8 Examples of Policy Bases

In this section, we present two examples of using policy bases to specify authorization requirements. The first example is the Bell-LaPadula model (BLP) [3]. We present a straightforward formulation of the basic BLP model in the policy base notation and also an enhancement with *need-to-know* restrictions. The second example shows how to formalize inheritance properties (as illustrated by examples in Section 2).

The essence of the basic BLP model can be summarized by two rules, "no read up and no write down". To simplify our presentation, we consider only two security levels low and high. We specify the BLP model as follows:

$$BLP = R^- \cup W^- \cup R^+ \cup W^+$$

where

$$\begin{array}{lrcl} \text{(No Read Up)} & R^- & = & \{s \in \mathsf{low} \wedge o \in \mathsf{high} \Rightarrow \mathsf{read}^-(s,o)\} \\ \text{(No Write Down)} & W^- & = & \{s \in \mathsf{high} \wedge o \in \mathsf{low} \Rightarrow \mathsf{write}^-(s,o)\} \\ \text{(Can Read Down)} & R^+ & = & \{o \in \mathsf{low} \Rightarrow \mathsf{read}^+(s,o)\} \\ \text{(Can Write Up)} & W^+ & = & \{s \in \mathsf{low} \Rightarrow \mathsf{write}^+(s,o)\} \end{array}$$

In the above, denials are absolute in the sense that no exception is allowed. Given a complete description of the group relation, the above policy base uniquely defines a strongly sound and strongly complete authorization policy that satisfies the *simple* and $\star$-security properties [3].

However, this basic model suffers from two drawbacks. First, the group relation must be completely defined in order to give a strongly complete authorization policy. Second, although positive authorizations that are granted do satisfy the simple and $\star$-security property, they violate the *principle of minimal privileges* [33].

We remedy this by adding need-to-know restrictions and denials by default. We modify R^+ and W^+ to be (respectively)

$$\begin{array}{rcl} R' & = & \{o \in \mathsf{low} \wedge \mathsf{need\text{-}to\text{-}know}^+(s,o) \Rightarrow \mathsf{read}^+(s,o)\} \\ W' & = & \{s \in \mathsf{low} \wedge \mathsf{need\text{-}to\text{-}know}^+(s,o) \Rightarrow \mathsf{write}^+(s,o)\} \end{array}$$

and BLP to be

$$BLP' = R^- \cup W^- \cup R' \cup W' \cup D$$

where D is

$$\left\{ \frac{:\ \mathsf{read}^-(s,o)}{\mathsf{read}^-(s,o)},\ \frac{:\ \mathsf{write}^-(s,o)}{\mathsf{write}^-(s,o)},\ \frac{:\ \neg\mathsf{need\text{-}to\text{-}know}^+(s,o)}{\neg\mathsf{need\text{-}to\text{-}know}^+(s,o)} \right\}$$

The virtual right need-to-know formalizes the need-to-know restrictions and can be defined in terms of *compartments* of subjects and objects using other rules.

We now turn to our second example. Consider the following inheritance properties:

> *(1) If a subject s has not been explicitly granted a right r to an object o, then s will inherit a denial of r to o if it belongs to a group g that has a denial of r to o. (2) If a subject s has not been explicitly denied a right r to an object o, then s will inherit a grant of r to o if all groups to which s belongs have grants of r to o.*

These can be expressed respectively by the following schemas:

$$d_1 = \frac{s \in g\ \wedge\ r^-(g,o)\ :\ \neg r^+(s,o)\ \wedge\ r^-(s,o)}{r^-(s,o)}$$

and

$$d_2 = \frac{\forall\, g[\neg(s \in g)\ \vee\ r^+(g,o)]\ :\ r^+(s,o)\ \wedge\ \neg r^-(s,o)}{r^+(s,o)}$$

where $\forall\, g[f(s,g,o)]$ in d_2 is a shorthand for the conjunction of all formulas of the form $f(s,\mathsf{G},o)$ where $\mathsf{G} \in \mathcal{S}$.

9 Implementation Considerations

Our model can be implemented as follows in a distributed system. Each policy base is stored and managed by a node in the system. We call such a node a *policy server*. These policy servers are organized in a hierarchical manner. Policy servers at the same level are called *peers*. Clients submit their access requests to appropriate policy servers for authorization decisions. The policy servers communicate with each other in authorizing an access request.

The assignment function used in interpreting propositional variables is implemented by a set of *distributed monitors* that keep track of the status of propositional variables.

The group relation is implemented by a set of *group servers* that collectively maintain group membership information for all subjects and objects in the system. Thus all updates of group memberships (e.g., additions and deletions) in the system are handled by the group servers.

Both the distributed monitors and group servers are regularly queried by the policy servers in making authorization decisions. The evaluation mechanism used in each policy server is based on an interpreter for general logic programs. In fact, a suitably modified Prolog interpreter is sufficient.

A preliminary design of a distributed authorization service based on the ideas presented in this paper is given in [38].

10 Concluding Remarks

We have presented a new approach to representing and evaluating authorization. In our approach, a set of authorization requirements is specified declaratively by a policy base. Unlike most existing approaches, the semantics of authorization is defined independently and is separate from implementation mechanisms.

Our approach is readily extensible. New predicate symbols can be added to our representation language to increase its expressiveness without a significant increase in computational requirements.

The existence of multiple authorities can be modeled in our approach by the use of suitable *composition* operators. Our research suggests that there are two notions of composition for policy bases that are important in a distributed system environment.

First, a system may be administered by multiple security administrators, each responsible for a distinct part of the system. Each security administrator specifies a policy base for the part of the system he is responsible for. In this case the different policy bases complement each other, in the sense that each fills in a part that has not been specified by others. Thus a composition gives the "sum" of all authorization requirements in the policy bases. We call this type of composition *peer* or *horizontal* composition.

Second, a security administrator may delegate his responsibilities to a number of subordinate administrators. This gives rise to a *root* policy base corresponding to the delegating administrator and a number of *leaf* policy bases corresponding to the subordinate administrators. The leaf policy bases are more specific and detailed than the root policy base and typically contain refinements of the root policy base. Composition in this case would combine all of the authorizations present in the root policy base together with their refinements in the leaf policy bases. We call this type of composition *hierarchical* or *vertical* composition.

The key difference between horizontal and vertical compositions is in their resolution of conflicts. A formal definition of these operators and their properties are still under investigation. Some preliminary ideas have been given in [37].

We are building a prototype implementation of the ideas in this paper. As we learn from our implementation experience, we may further refine our language for pragmatic or efficiency consideration. Specifically, we may restrict the language's expressive power. Such interplay between efficiency and expressiveness is an important area for future research.

Lastly, there are some other general problems that deserve further investigation. These include: (1) the use of disjunctive information in authorization, and (2) the incorporation of structured subjects (e.g., one subject being a role or delegate of another subject) and structured objects (e.g., one object being an implementation of another object) into our representation language.

Acknowledgements

The authors would like to thank Clifford Neuman and the anonymous referees for their helpful comments on revising the paper. The authors are also grateful to Vladimir Lifschitz for several useful discussions on our paraconsistent semantics for extended logic programs.

References

[1] M. Abadi, M. Burrows, B.W. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993. An abbreviated version appeared in Advances in Cryptology — CRYPTO '91, pages 1–23, Santa Barbara, California, August 11–15 1991.

[2] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowlege. In J. Minker, editor, *Foundations of Deductive Database and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., 1988.

[3] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, March 1976.

[4] P. Bieber and F. Cuppens. A logical view of secure dependencies. *Journal of Computer Security*, 1(1):99–129, 1992.

[5] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 45–54, Asilomar Conference Grounds, Pacific Grove, California, December 10-12 1979.

[6] T.A. Budd. Safety in grammatical protection systems. *International Journal of Computer and Information Sciences*, 12(6):413–431, 1983.

[7] D.E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[8] D.E. Denning, T.F. Lunt, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView formal security policy model. Technical Report A003, Computer Science Laboratory, SRI International, 1987.

[9] M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold Company, New York, 1988.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference*, pages 1070–1080. The MIT Press, 1988.

[11] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3,4):365–385, 1991.

[12] J. Glasgow, G. MacEwen, and P. Panangaden. A logic for reasoning about security. In *Proceedings of the The Computer Security Foundations Workshop III*, pages 2–13, Franconia, New Hampshire, June 12–14 1990.

[13] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, California, April 26–28 1982.

[14] G.S. Graham and P.J. Denning. Protection — principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429, Atlantic City, New Jersey, May 16–18 1972.

[15] P.P. Griffiths and B.W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.

[16] M.A. Harrison. Theoretical issues concerning protection in operating systems. In M.C. Yovits, editor, *Advances in Computers*, volume 24, pages 61–100. Academic Press, 1985.

[17] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[18] A.K. Jones. Protection mechanisms and the enforcement of security policies. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems An Advanced Course*, chapter 3.C, pages 228–251. Springer Verlag, 1979.

[19] S. Kramer. On incorporating access control lists into the Unix operating system. In *Proceedings of the Usenix Unix Security Workshop*, pages 38–48, Portland, Oregon, August 29-30 1988.

[20] B.W. Lampson. Dynamic protection structures. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 35, pages 27–38, Las Vegas, Nevada, November 18–20 1969.

[21] B.W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in ACM Operating Systems Review, 8(1):18–24, January 1974.

[22] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[23] C.E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[24] C.E. Landwehr, C.L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Transactions on Computer Systems*, 2(3):198–222, August 1984.

[25] R.J. Lipton and T.A. Budd. On classes of protection system. In R.A. DeMillo, D.P. Dobkin, A.K. Jones, and R.J. Lipton, editors, *Foundations of Secure Computations*, pages 281–296. Academic Press, 1978.

[26] T.F. Lunt. Access control policies: Some unanswered questions. *Computer & Security*, 8(1):43–54, February 1989.

[27] J. McLean. The algebra of security. In *Proceedings of the 9th IEEE Symposium on Research in Security and Privacy*, pages 2–7, Oakland, California, April 18–21 1988.

[28] J. McLean. Security models and information flow. In *Proceedings of the 11th IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 7–9 1990.

[29] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

[30] J.K. Millen. Models of multilevel computer security. In M.C. Yovits, editor, *Advances in Computers*, volume 29, pages 1–45. Academic Press, 1989.

[31] T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, June 1989.

[32] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, April 1980.

[33] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[34] R.S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432, April 1988.

[35] R.S. Sandhu. Expressive power of the schematic protection model. *Journal of Computer Security*, 1(1):59–98, 1992.

[36] L. Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, C-30(3):172–181, March 1981.

[37] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 13th IEEE Symposium on Research in Security and Privacy*, pages 33–50, Oakland, California, May 4–6 1992.

[38] T.Y.C. Woo and S.S. Lam. A framework for distributed authorization (extended abstract). In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 112–118, Fairfax, Virginia, November 3–5 1993.

A Proof of Theorem A

We first observe some simple lemmas.

Lemma 1. Let $\mathcal{I}$ be an assignment, $\mathcal{G}$ a group relation, Σ a set of distinguished literals and f a formula. Then

$$\Sigma \models_{\mathcal{I},\mathcal{G}} f \text{ iff } \Sigma \models [\mathcal{I},\mathcal{G}](f)$$

Proof. Immediate from the definition of $\models_{\mathcal{I},\mathcal{G}}$. Note that $[\mathcal{I},\mathcal{G}](f)$ is a pure formula. □

Lemma 2. Let $\mathcal{I}$ be an assignment, $\mathcal{G}$ a group relation, Σ a set of distinguished literals and f a formula. Then

$$S^{\mathcal{I},\mathcal{G}}_{B,\Sigma} = S_{[\mathcal{I},\mathcal{G}](B),\Sigma}$$

Proof. Note that $[\mathcal{I},\mathcal{G}](B)$ is a pure base. Let M be a set of distinguished literals.

$M \in S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$

iff { definition of $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$ }

for all $\frac{f\,:\,f'}{g} \in B$, if $M \models_{\mathcal{I},\mathcal{G}} f$ and $\Sigma \not\models neg(f')$ then $M \models g$

iff { Lemma 1, f' and g are basic formulas }

for all $\frac{f\,:\,f'}{g} \in B$, if $M \models [\mathcal{I},\mathcal{G}](f)$ and $\Sigma \not\models [\mathcal{I},\mathcal{G}](neg(f'))$ then $M \models [\mathcal{I},\mathcal{G}](g)$

iff { replacing $[\mathcal{I},\mathcal{G}](h)$ by h and definition of $[\mathcal{I},\mathcal{G}](B)$ }

for all $\frac{f\,:\,f'}{g} \in [\mathcal{I},\mathcal{G}](B)$, if $M \models f$ and $\Sigma \not\models neg(f')$ then $M \models g$

iff { definition of $S_{[\mathcal{I},\mathcal{G}](B),\Sigma}$ }

$M \in S_{[\mathcal{I},\mathcal{G}](B),\Sigma}$

□

Theorem A. Let B be a base, $\mathcal{I}$ an assignment and $\mathcal{G}$ a group relation. Let Σ be a set of distinguished literals. Then

Σ is an extension of B under $\mathcal{I}$ and $\mathcal{G}$ iff Σ is an extension of $[\mathcal{I},\mathcal{G}](B)$

Proof.

Σ is an extension of B under $\mathcal{I}$ and $\mathcal{G}$

iff { definition of extension }

$\Sigma = \Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma)$

iff { definition of $\Gamma_{B,\mathcal{I},\mathcal{G}}(\Sigma)$ }

$\Sigma =$ the intersection of all elements in $S^{\mathcal{I},\mathcal{G}}_{B,\Sigma}$

iff { Lemma 2 }

$\Sigma =$ the intersection of all elements in $S_{[\mathcal{I},\mathcal{G}](B),\Sigma}$

iff { definition of $\Gamma_{[\mathcal{I},\mathcal{G}](B)}(\Sigma)$ }

$\Sigma = \Gamma_{[\mathcal{I},\mathcal{G}](B)}(\Sigma)$

iff { definition of extension }

Σ is an extension of $[\mathcal{I},\mathcal{G}](B)$

□

B A Paraconsistent Semantics for Extended Logic Programs

In the following, a literal refers to a distinguished literal.

B.1 Programs

A *program formula* (or *formula* in short) is:

- T or F
- a literal
- $f \wedge f'$ where f and f' are formulas
- $f \vee f'$ where f and f' are formulas

Note that negation occurs only at the literal level, and not at the formula level. A *conjunctive* formula is a formula without disjunction.

A *program clause* (or *clause* in short) is $f \leftarrow g$ where f is a conjunctive formula and g a formula. g is called the *premise* while f the *consequence* of the rule. A *closed rule* is a rule that does not have any free variables. A *program* is a set of clauses. A *closed program* is a program containing only closed clauses. We consider only closed clauses and closed programs in the sequel. Thus, all mentions of clauses and programs are assumed to be closed.

We want to define the concept of a *model* for a program. To do this, we first define a *satisfaction* relationship between a set of literals and a formula or clause.

Let Σ be a set of literals. We have:

- $\Sigma \models \mathsf{T}$
- $\Sigma \not\models \mathsf{F}$
- $\Sigma \models L$ iff $L \in \Sigma$ where L is a literal
- $\Sigma \models f \wedge f'$ iff $\Sigma \models f$ and $\Sigma \models f'$
- $\Sigma \models f \vee f'$ iff $\Sigma \models f$ or $\Sigma \models f'$
- $\Sigma \models f \leftarrow g$ iff $\Sigma \models g$ implies $\Sigma \models f$

We observe that the above definition for $\models$ when restricted to program formulas is identical to the definition of $\models_{\mathcal{I},\mathcal{G}}$ for formulas without ordinary literals.

Definition. Let Π be a program. Σ is a *model* of Π iff for all clause $r \in \Pi$, $\Sigma \models r$. □

Note that the set of all literals is always a model of any positive program Π.

Proposition. Let Π be a program. Then the intersection of all models of Π is also a model of Π. $\Box$

Corollary. For any program Π, there exists a least model of Π. $\Box$

We denote the least model of program Π by M_Π. Thus, $M_\Pi =$ is the smallest set Σ such that for each clause $r \in \Pi$, $\Sigma \models r$. Note that M_Π is computable by a "bottom up" evaluation. More precisely, let $\Sigma_0 = \emptyset$, and let Σ_{i+1} be the smallest set such that (1) it includes Σ_i and (2) for every clause $f \leftarrow g \in \Pi$, if $\Sigma_i \models g$ then $\Sigma_{i+1} \models f$. If there exists an i such that $\Sigma_j = \Sigma_i$, for all $j \geq i$, then $M_\Pi = \Sigma_i$.

B.2 Extended Programs

An *extended literal* is L or not L where L is a literal. An *extended program clause* (or *extended clause* in short) is constructed exactly like a clause except that extended literals are allowed in the premise of the rule in place of literals. An *extended program* is a set of extended clauses.

We want to define an analogous notion of model for an extended program. To do that, we first define a reduction from an extended program to a program. Let Σ be a set of literals and r an extended clause. We define an operation Θ_Σ that transforms r into a regular clause. The definition of Θ_Σ is as follows:

- replace all occurrences of not T in r with F,
- replace all occurrences of not F in r with T,
- replace all occurrences of not L in r with F if $L \in \Sigma$,
- replace all occurrences of not L in r with T if $L \notin \Sigma$.

The resulting clause is denoted by $\Theta_\Sigma[r]$. Note that since extended literals are only allowed in the premise of a clause, so Θ_Σ can be viewed as an operation on formulas instead of clauses. This view will be used later on.

Let Π^Σ denote the program obtained from an extended program Π by applying Θ_Σ to each clause in Π. That is, $\Pi^\Sigma = \{\Theta_\Sigma[r] \mid r \in \Pi\}$.

Definition. Σ is a *model* of Π iff $\Sigma = M_{\Pi^\Sigma}$. $\Box$

While an extended program may have zero, one or multiple models, there are syntactic characterizations of extended programs that admit unique models. For example, the existence of a stratification [2] is one such characterization.

C Proof of Theorem B

In the following, a literal refers to a distinguished literal. Note also that a program formula has the same syntax as a formula in a pure rule. Hence in the following, a formula is interpreted according to its context.

Lemma 3. Let Σ be a set of literals and f be a formula. Then

$$\Sigma \not\models f \text{ iff } \Theta_\Sigma[not(f)] \equiv \mathsf{T}$$

where $\equiv$ denotes the usual logical equivalence.

Proof. By induction on the structure of formulas. □

Lemma 4. Let Σ be a set of literals and f be a formula. Then for any set M of literals

$$M \models \Theta_\Sigma[not(f)] \text{ iff } \Theta_\Sigma[not(f)] \equiv \mathsf{T}$$

Proof. By induction on the structure of formulas. □

Lemma 5. Let B be a pure base. Then for any set Σ of literals

$$\Gamma_B(\Sigma) = M_{(\pi B)^\Sigma}$$

Proof. Before we begin the proof, we make the following observation: By definition, all clauses in $(\pi B)^\Sigma$ are obtained from the clauses in πB. In particular, we have:

$$r \in \pi B \text{ iff } \Theta_\Sigma[r] \in (\pi B)^\Sigma$$

Now since πB is the extended program obtained by translation from base B, all clauses in πB are of the following form

$$g \leftarrow f \wedge not(neg(f'))$$

Hence, all clauses in $(\pi B)^\Sigma$ are of the form

$$g \leftarrow \Theta_\Sigma[f \wedge not(neg(f'))]$$

which simplifies to

$$g \leftarrow f \wedge \Theta_\Sigma[not(neg(f'))]$$

as both f and g do not contain extended literals.

To summarize, we have

$$\begin{array}{lll} \frac{f\,:\,f'}{g} \in B & \text{iff} & g \leftarrow f \wedge not(neg(f')) \in \pi B \\ & \text{iff} & g \leftarrow f \wedge \Theta_\Sigma[not(neg(f'))] \in (\pi B)^\Sigma \end{array} \qquad (*)$$

Now back to the proof, we prove the equality by showing subset inclusion in both directions.

(a) $\Gamma_B(\Sigma) \subseteq M_{(\pi B)^\Sigma}$.

If we can prove $M_{(\pi B)^\Sigma} \in S_{B,\Sigma}$, then by the fact that $\Gamma_B(\Sigma)$ is the least element of $S_{B,\Sigma}$, we obtain the desired inclusion. In the following, we prove $M_{(\pi B)^\Sigma} \in S_{B,\Sigma}$ by showing it satisfies the condition for being an element of $S_{B,\Sigma}$.
First, from the fact that $M_{(\pi B)^\Sigma}$ is the least model of $(\pi B)^\Sigma$, we have:

For all clause $g \leftarrow f \wedge \Theta_\Sigma[not(neg(f'))] \in (\pi B)^\Sigma$,

$$M_{(\pi B)^\Sigma} \models f \text{ and } M_{(\pi B)^\Sigma} \models \Theta_\Sigma[not(neg(f'))] \text{ implies } M_{(\pi B)^\Sigma} \models g$$

By $(*)$ and Lemma 4, we get:

For all rule $\frac{f\,:\,f'}{g} \in B$,

$$M_{(\pi B)^\Sigma} \models f \text{ and } \Theta_\Sigma[not(neg(f'))] \equiv \mathsf{T} \text{ implies } M_{(\pi B)^\Sigma} \models g$$

Then by Lemma 3, we get:

For all rule $\frac{f\,:\,f'}{g} \in B$,

$$M_{(\pi B)^\Sigma} \models f \text{ and } \Sigma \not\models neg(f') \text{ implies } M_{(\pi B)^\Sigma} \models g$$

which is exactly the condition for being an element of $S_{B,\Sigma}$.

(b) $M_{(\pi B)^\Sigma} \subseteq \Gamma_B(\Sigma)$.

If we can prove that $\Gamma_B(\Sigma)$ is a model of $(\pi B)^\Sigma$, then by the fact that $M_{(\pi B)^\Sigma}$ is the least model of $(\pi B)^\Sigma$, we obtain the desired inclusion. In the following, we show that $\Gamma_B(\Sigma)$ satisfies the condition for a model of $(\pi B)^\Sigma$.
First, since $\Gamma_B(\Sigma) \in S_{B,\Sigma}$, we have:

For all rule $\frac{f\,:\,f'}{g} \in B$,

$$\Gamma_B(\Sigma) \models f \text{ and } \Sigma \not\models neg(f') \text{ implies } \Gamma_B(\Sigma) \models g$$

By Lemma 3, we get:

For all rule $\frac{f\,:\,f'}{g} \in B$,

$$\Gamma_B(\Sigma) \models f \text{ and } \Theta_\Sigma[not(neg(f'))] \;\equiv\; \mathsf{T} \text{ implies } \Gamma_B(\Sigma) \models g$$

By $(*)$ and Lemma 4, we get:

For all clause $g \;\leftarrow\; f \;\wedge\; \Theta_\Sigma[not(neg(f'))] \in (\pi B)^\Sigma$,

$$\Gamma_B(\Sigma) \models f \text{ and } \Gamma_B(\Sigma) \models \Theta_\Sigma[not(neg(f'))] \text{ implies } \Gamma_B(\Sigma) \models g$$

which is the condition for $\Gamma_B(\Sigma)$ to be a model of $(\pi B)^\Sigma$. □

Theorem B. Let B be a pure base and Σ a set of literals. Then

Σ is an extension of B iff Σ is a model of πB

Proof.

$$\begin{array}{ll} & \Sigma \text{ is an extension of } B \\ \text{iff} & \{ \text{ definition of extension } \} \\ & \Sigma = \Gamma_B(\Sigma) \\ \text{iff} & \{ \text{ Lemma 5 } \} \\ & \Sigma = M_{(\pi B)^\Sigma} \\ \text{iff} & \{ \text{ definition of model } \} \\ & \Sigma \text{ is a model of } \pi B \end{array}$$

□