1. Logics for Authorizations and Security

Piero A. Bonatti and Pierangela Samarati

Dipartimento di Tecnologie dell'Informazione, Università di Milano
Via Bramante 65, I-26013 Crema, Italy
{bonatti,samarati}@dti.unimi.it

Today's computer systems and applications are based on a Globally Internetworked Infrastructure, connecting remote parties through large scale networks such as the World Wide Web. The network-based nature of applications improves information availability, but at the same time it increases system vulnerability, calling for powerful and reliable tools for specifying and enforcing security properties.

The expressive power and the formal foundations of logical formalisms are appealing in this context. Indeed, logics of various kinds have already been investigated and are being proposed as suitable means for tackling many facets of security.

Some researchers and practitioners object that logic-based specifications may result complicated or even intimidating to some users. Security administrators and end users need simple and friendly approaches that allow them to easily understand the system behavior and maintain control over security specifications. It is tempting to conclude that logic-based approaches are not applicable, but then one would also give up all the advantages of logic-based formulations that enjoy a combination of clean foundations (hence formal guarantees), flexibility, expressiveness, and declarativeness (so that users are not required any programming ability). On the contrary, we believe that a careful choice of syntax makes logic-based specifications accessible to a wide spectrum of users. For example, formulae may be represented graphically [1.80, 1.21], and syntax can be made similar to natural language [1.80, 1.20]. Behind such nonstandard syntax we still find logic and all of its advantages.

While an informal definition of security can be straightforwardly given as "protecting information and resources from unauthorized or improper use, and from being damaged", a precise specification is a subtle and difficult matter. At least the following issues should be addressed:

- What is to be protected and from whom/what? and, more importantly, what does it *mean* to be protected? This is one of the most delicate aspects. The above questions, in general, do not have a unique (or obvious) answer and the properties characterizing security may be different for different applications. The formalization language should be rich enough to model all the relevant aspects for the given scenario.
- How do we *specify* what is authorized or forbidden? The decision may depend on different dimensions such as properties of the parties (e.g., role in the organization) or time and context of the access. Because of the open nature of modern systems, specifications must be expressive enough to cover

a vast number of possible situations. Moreover, to keep system security under control, specifications should be concise, precise and unambiguous, and it should be possible to formulate them modularly and incrementally. At the same time, specifications must be enforceable efficiently, and there is a trade-off between expressiveness requirements and implementability.
– How can we *prove* that systems/applications are secure with respect to the chosen meaning of security? This question calls for formal description of the system behavior and (possibly automated) methods for proving properties of it.

The above points have been tackled using a variety of logics. For instance, in order to define what it means for a system to be secure, sophisticated logics (such as multi-modal logics) have been used to model aspects such as user beliefs and inference abilities, interaction histories, and so on. Various computational logics have been used as policy specification languages. These include Horn clauses extended with inheritance, temporal constraints, negation as failure, and other features. Finally, the verification of security properties may be based on automated reasoning techniques such as theorem proving and SAT procedures.

The proposed solutions provide satisfactory answers to classical security problems. However, as we move to interactions between different, remotely located parties that may know little about each other, new ways of specifying and enforcing security have to be adopted. For instance, answering "yes" or "no" to an access request may not be sufficient anymore, and the system may need to communicate the conditions under which access can be granted. As a consequence some form of negotiation may arise.

In this chapter we survey some of the major applications of logics to the specification, verification and enforcement of security policies, with special attention to the security issues that arise in open and distributed systems. We shall point out some practical aspects that may increase the usability of logic-based techniques. Finally, we shall identify some possible directions for further research on security, and suggest some possible contributions of logics and automated reasoning in this context.

Terminology—and especially the word "policy"—is used in the literature with subtly different meanings. In this chapter we draw a clear distinction between two conceptually different aspects: (i) how to formulate the set of authorizations that belong to a policy, and (ii) the meaning of these authorizations, that is, their effect on the behavior of the system and their relationships with the security goals to be achieved. In this chapter we focus on the former aspect, and this will lead us to focus mainly on logic programming techniques, that underly most approaches to the specification of authorization sets. There will be less emphasis on multimodal logics, including [1.3], that have been mostly applied to aspect (ii).

We mainly address two kinds of reader. For the reader interested in what logics can do for security, we informally illustrate policy specification languages and their expressiveness. For the reader with a background in computer science logics, we hint at the technical problems tackled by different approaches and expand only in those cases that induced extensions to the standard theory.

The chapter is organized as follows. After a preliminary section on the structure of common formalizations of security, in Section 1.2 we illustrate a number of logic-based languages for expressing and enforcing security policies. This section is the core of the chapter, and illustrates constructs to formulate dynamic policies, inheritance and overriding, message control policies, policy composition, and credential-based authorizations. Section 1.3 (addressed to the reader with background on logic programming and non monotonic reasoning) illustrates some new non monotonic inference mechanisms motivated by policy verification. Section 1.4 sketches other important applications of logic to security. Finally, Section 1.5 draws our conclusions and highlights some interesting directions for further research.

1.1 Preliminaries on security formalizations

In general, a logical formalization of security consists of three components (a similar view can be found in [1.28]):

1. *An abstract computational model of the system.*
   The computational model may range from a Turing-equivalent model, such as abstract state machines [1.36], to something simpler as a game tree or traces that correspond to branches of a game tree [1.79, 1.15, 1.13]. The computational model determines the aspects of the system that can be verified w.r.t. security. The formalization can only deal with what is observable through the computational model.
   On one hand, this is the main limitation of the formalizations of security. Typically, security threats exploit features that are not captured by the computational model, such as operating system bugs, and flaws in the authentication procedure. As long as they are not captured in the computational model, formal analysis cannot detect them. Moreover, real systems are complex enough to make a faithful formal computational model impossible in practice.
   On the other hand, the computational model describes (more or less explicitly) the set of assumptions on the system that are needed for the security proofs. In other words, the computational model can be used to encode the prerequisites that the system should satisfy in order to enjoy the desired security properties.
2. *A security policy.*
   Security policies consist of sets of constraints on the system behavior, such as the set of actions that each user is allowed to invoke, or the information that each user is allowed to receive. Clearly, the language in

which policies are expressed and its semantics depend on the security properties that must be enforced. Frequently, the constraints in a policy are *authorizations* of the form (*subject,action,object*). Intuitively such a triple means that the given subject is authorized to perform the specified operation on the given object. A classical access control matrix is a particular representation of a policy, which can then be implemented as a table indexed by subject (access lists), by object (capability lists), or as catalogs (authorization tables) [1.75]. In many cases, authorizations may change along time. In this case, authorizations are extended with an element representing time (e.g., [1.28]). In some settings, authorizations are further extended, e.g., with the subject who granted the authorization, or with a sign for *negative* authorizations (e.g., [1.83]) sometimes called *denials*. In other approaches, some of the fields of authorization triples may be implicit or even missing. For instance, the authorizations may tell what each subject is allowed to *know*, and for this purpose, the action field can be omitted [1.79, 1.13, 1.28, 1.15].

In their simplest form, policies can be expressed extensionally, as sets of authorizations. However, such a flat formulation is not suitable in practice because real world policies are becoming increasingly complex. We defer the discussion of the factors that increase policy complexity until the next section. Here it suffices to say that in order to dominate policy complexity and reduce the probability of overlooking important details, significant efforts are being focused on flexible and modular policy definition languages. The experience gathered in the area of declarative programming languages is being applied in this area.

When authorizations are specified implicitly through a set of logical axioms, access control involves some kind of automated deduction. In this chapter we shall mostly focus on this particular application of logics to security.

3. *A formal definition of one or more security properties.*

   The security properties specify the security goals to be achieved (e.g., "There must be no information flow from user A to user B").

   Note the conceptual difference between security properties and security policies, as defined in our terminology. Policies are just *syntactic* representations of security constraints. The *meaning* of the constraints is specified elsewhere, as explained below.

   In some contexts, constraints are directly interpreted in terms of the system's behavior. For example, an authorization (subject,object,operation) typically means that the system will perform the given operation on the given object when the operation is requested by the given user. In such frameworks, the policy can be regarded as a particular implementation of the specifications formulated by the security properties. Hopefully, the policy suffices to achieve the security properties.

In other contexts, the meaning of the constraints is specified by the security properties themselves. For example, a set of system variables P has no particular meaning in itself. The meaning of P as a policy may be specified by a security property stating—say— that a user A that can read the values of the variables listed in P should not be able to infer the operations performed by another user B on the system. In this example, the system's behavior is not influenced by the policy. The policy is rather used as an "input" to the security property, to make it parametric with respect to the set of variables P.

Of course, in some contexts, the constraints may both influence the system behavior *and* concur to the formulation of the security properties to be achieved.

The abstract model of the system should satisfy the security properties for all possible policies expressed in the adopted language (triples, sets of variables, etc.). Typical security properties pose restrictions on information flow and on user knowledge of the system's data. For example, *secrecy-based multilevel models* prescribe that information must only flow from low to high levels in a partially ordered set of security levels [1.73]. Other properties such as *non-deducibility, non-interference* and *causality* (e.g., [1.36, 1.11]) state in different ways that no user U should be able to infer the behavior of any other user W (here, by "behavior" we mean the sequence of operations invoked on the information system). Further examples are given by the notions of security related to data confidentiality adopted in [1.79, 1.12, 1.13]. Almost all of the aforementioned security properties are based on some form of *undistinguishability*, that is, the details of the system behavior cannot be derived from the observables. Roughly speaking, the projection of the system onto the observables should not be invertible. The observables are determined by the policy.

For the readers with background in modal logics, we note that the abstract computational model of the system can usually be presented as a Kripke structure, where the accessibility relation represents a temporal modality. Such interpretation structures can be extended by adding accessibility relations representing user knowledge [1.11, 1.39] or user beliefs [1.15] and—sometimes—deontic modalities to state what is permitted (modality P, sometimes denoted by R), forbidden (F), obligatory (O) [1.11, 1.35, 1.39, 1.28, 1.29, 1.30]. Deontic modalities can then be used to formulate security policies [1.29].

1.2 Logic-based policy specification languages

The size and the complexity of real world security policies makes it impossible to handle "flat" policy representations such as plain authorization lists.

Policy complexity is increased by the interplay of multiple, heterogeneous requirements, that must be harmonized and merged into a coherent policy.

- In complex organizations, different branches or departments may have direct control over their own data, and establish their own security policy. At the organization level, interoperability of all these information subsystems is required, and their different policies must be merged.
- Frequently, different organizations share part of their data, and must agree on a disclosure policy. For example, national statistical institutes typically do not own the data they distribute—data collectors retain data ownership. So the distributor and the owner must agree on a security policy, merging the requirements of both.
- National laws are another *external* source of security constraints. In particular, privacy laws (that are currently enforced in different forms by many countries) require sensitive personal data to be protected from any use not explicitly authorized by the owner.
- Even within a single, homogeneous system, different groups of users and different classes of objects may be treated according to different principles.
- A related issue is that there is a trade off between protection and system usability, especially when the system is open to the Internet. One of the most famous examples is the security model of Java. The initial strict sandbox model soon turned out to be too rigid for the intended applications of Java, and a finer grained model, based on a more expressive policy language, has been adopted for Java 2 [1.38].

A further degree of complexity is introduced by the temporal dimension. The authorizations granted by a policy may change along time. So the specification language should be able to express multiple access control policies and the dynamic behavior of the policy within a single framework.

The approaches that focus on policy specification languages usually leave the abstract computational model and the security properties implicit, and concentrate on language constructs that enhance clarity, modularity, and other desirable properties. The semantics of policy languages determine the *extension* of each policy (i.e., the set of authorizations granted by the policy), rather than the meaning of authorizations (i.e., the intended *effect* of authorizations on the system's behavior, which is determined by the security properties, cf. point 3 in Section 1.1).

Logic languages are particularly attractive as policy specification languages. One obvious advantage lies in their clean and unambiguous semantics, suitable for implementation validation, as well as formal policy verification [1.21, 1.59]. Second, logic languages can be expressive enough to formulate all the policies introduced in the literature. The declarative nature of logic languages yields a good compromise between expressiveness and simplicity. Their high level of abstraction, very close to the natural language formulation of the policies, makes them simpler to use than imperative programming languages, especially for people with little or no technical training (such as typical security managers). However, such people are not experts in formal logics, either, so generality is sometimes traded for simplicity. For this reason, some languages do not adopt a first-order syntax, even if the policy language is then interpreted by embedding it into a first-order logic (e.g., [1.7, 1.21]).[1]

The embedding often isolates a fragment of the target logic with nice computational properties. We have already pointed out that efficiency is an issue. In a real system with hundreds of users and hundreds or thousands of data objects, the set of potential authorizations may have 10^4-10^5 elements or more. Moreover, if the policy is time-dependent, then the number of authorizations may increase significantly. Therefore, one can only afford policy languages with low polynomial complexity. In fact, most of the logic-based policy specification languages proposed so far are directly or indirectly mapped onto more or less extended forms of logic programs, suitable for efficient, PTIME implementations. Moreover, the implementations tend to *materialize* policy extensions (i.e., the canonical model of the logic program), in order to speed-up the system response. There should be no inference at runtime.

The target logic is typically *non monotonic*, that is, the set of consequences of a theory[2] does not increase monotonically with the set of axioms in the theory. Policy specification (beyond the realm of security) has been proposed long ago as an application of non monotonic logics [1.58]. The reason is that sometimes decisions must be made in the absence of information. So, when new information is added to the theory, some decision may have to be *retracted* (because they have lost their justification), thereby inducing a non monotonic behavior. In the area of security, such default decisions arise naturally in real world policies. For example, *open* policies, prescribe that by default authorizations are granted, while *closed* policies prescribes that they should be denied unless stated otherwise. It will be shown later that a particular form of non monotonic reasoning (inheritance with overriding) is useful for incremental and compact policy specification.[3]

It is well known that logic programs with negation-as-failure can be regarded as a fragment of major non monotonic logics such as *default logic* [1.68] and *autoepistemic logic* [1.64], provided that negation-as-failure is interpreted under the *stable model semantics* [1.34]. This semantics maps each logic program onto a set of minimal models called *stable models* [1.34]. There are opposite points of view on this feature.

---

[1] Here, by "a first-order logic" we simply mean a logic with individual variables and quantifiers, be it modal, non monotonic, or simply a classical first-order logic.

[2] By "theory" we mean a set of formulae, sometimes called *axioms*.

[3] In some simple cases, non monotonic reasoning can be reduced to classical reasoning with Clarke's completion and related methods. However, these methods fail in the presence of recursive rules that are very frequent in security policies, especially when inheritance is to be modeled.

Some authors regard multiple models as an opportunity to write nondeterministic specifications, where each model is an acceptable policy, and the system makes an automatic choice between the available alternatives [1.10]. For instance, the models of a policy may correspond to all the possible ways of assigning permissions that preserve a *Chinese Wall* policy [1.22]. Intuitively, the Chinese wall policy is an interpretation of secrecy flow constraints in a discretionary context, where the access control system must ensure that users be not granted access to objects pertaining to companies which are in a mutual conflict of interest. The Chinese Wall principle dictates support for *dynamic separation of duty*, as the set of accesses that a user can actually execute is a subset of those for which she has been granted authorization and is determined dynamically (i.e., by exercising some privileges a user rules out her possibility of exercising other privileges). The system may then choose a secure permission assignment dynamically, trying to satisfy user requests. In terms of policy models, this corresponds to choosing one model by reacting to user requests. In general, the set of alternative models may grow exponentially, and the problem of finding one of them is NP-complete, but there are exceptions with polynomial complexity [1.70, 1.66].

Some other authors believe that security managers would not trust the system's automatic choice, and adopt restrictions such as *stratifiability* [1.5] to guarantee that the canonical model be unique. The system rejects non-stratified specifications, highlighting non-stratified rules to help the security administrator in reformulating the specifications.[4] As a further advantage, stratifiability-like restrictions yield PTIME semantics.

A non monotonic logic has been proposed for the first time as a policy specification language by Woo and Lam [1.83]. They show how default logic can be used to express a number of different policies. For example, if authorizations are expressed with a predicate $\texttt{auth}(\texttt{subj}, \texttt{action}, \texttt{obj})$ and $\top$ is a tautology, then the following default rules express open and closed policies, respectively.

$$\frac{\top : \texttt{auth}(x,y,z)}{\texttt{auth}(x,y,z)} \qquad \frac{\top : \neg\texttt{auth}(x,y,z)}{\neg\texttt{auth}(x,y,z)} \, .$$

Intuitively, the first rule says that if $\top$ is derivable (first premise) and it is consistent to assume $\texttt{auth}(x,y,z)$ (second premise, after the colon), then $\texttt{auth}(x,y,z)$ can be derived (conclusion). The second rule is similar.

Woo and Lam provide also an axiomatization of the Bell-LaPadula security model [1.6], refined with the *need-to-know* principle.

In order to address complexity issues (inference in default logic is at the second level of the polynomial hierarchy), Woo and Lam propose to use the fragment of default logic corresponding to *stratified, extended logic programs*, that is, stratified logic programs with two negations (negation as failure and

[4] Note that the Chinese Wall policy can be expressed also with stratified programs [1.45].

classical negation), whose unique stable model can be computed in quadratic time. Extended logic programs can be easily transformed into equivalent normal logic programs (with only negation-as-failure) by means of a straightforward predicate renaming.

Default logic is a very flexible policy specification language. Different users and objects can be treated with different policies. For example, open and closed policies may coexist if suitably restricted versions of the above rules—e.g., where $\top$ is replaced with some condition on x, y, z—are put together. In this way, the need of harmonizing heterogeneous requirements is taken into account.

The approach by Woo and Lam has been subsequently refined by several authors (e.g., [1.10, 1.45]). Some have proposed fixed sets of predicates and terms, tailored to the expression of security policies. In the language of the security community, such a fixed vocabulary is called a *model*, while in the AI community it would be probably regarded as an elementary ontology. From a practical point of view, the vocabulary guides security administrators in the specification of the policy. This is an example of how generality is often traded for simplicity.

Furthermore, the original approach has been extended with temporal constructs, inheritance and overriding, message control, policy composition constructs, and electronic credential handling. All these aspects are illustrated in detail in the following subsections.

1.2.1 Dynamic policies

Security policies may change along time. Users, objects and authorizations can be created and removed. Moreover, some authorizations may be active only periodically, e.g., an employee may use the system only during work hours. Therefore, policy languages should be able to express time-dependent behavior.

Policy modifications can be specified with an imperative language, but then security managers need some training in programming. A streamlined such language has been investigated by Harrison, Ruzzo and Ullman in [1.42]. Since an arbitrary number of users and objects can be created with the language, the policy extension has no fixed finite bound, and can encode the tape of an arbitrary Turing machine. Consequently, in this framework, the behavior of the policy extension is undecidable in general, although it can be decidable in restricted cases [1.43, 1.76, 1.77].

Temporal authorization bases In [1.7] the sets of users and objects are fixed, and the temporal validity of authorizations is specified through *periodic expressions* and suitable *temporal operators*. The resulting language is not completely general, but it has three major advantages: it is expressive enough for typical real-world policies, it does not require expertise in modal logic (actually, temporal expressions and operators are "hidden" behind natural-language-like expressions), and its inferences are decidable. In other words, the generality of temporal operators is traded for usability and decidability.

The temporal authorization model includes both periodic authorizations and temporal operators. Periodic authorizations are obtained by labeling each authorization—which is a 5-tuple of the form (*subject, object, action, sign, grantor*), where the sign determines whether the rule is an authorization or a denial—with a *temporal expression* specifying the time instants in which the authorization applies. Temporal expressions are formulated in a symbolic, user friendly formalism. They consist of pairs $\langle$`[begin,end]`,`P`$\rangle$. `P` is a *periodic expression* denoting an infinite set of time intervals (such as *"9 a.m. to 1 p.m. on Working-days"*). The temporal interval `[begin, end]` denotes the lower and upper bounds imposed on the scope of the periodic expression `P` (e.g., `[2/2002,8/2002]`). The authorization is valid in all time points that lie within the interval `[begin,end]` and satisfy the periodic expression `P`.

The policy specification language supports *derivation rules* that can be used to derive new authorizations from the presence or absence of other authorizations in specific periods of time. For instance, it can be specified that two users that work on the same project must receive the same authorizations on some given objects, or that a user should be authorized to access an object in a period P, only if nobody else was ever authorized to access the same object during P. The validity of a derivation rule can be restricted by labeling the rule with a temporal expression, by analogy with periodic authorizations. Formally, a derivation rule is a triple `([begin,end], P, A` $\langle$OP$\rangle$ $\mathcal{A}$`)`, where $\langle$`[begin,end]`,`P`$\rangle$ is the temporal expression, `A` is the (ground) authorization to be derived, $\mathcal{A}$ is a boolean composition of (ground) authorizations, and OP is one of the following operators: WHENEVER, ASLONGAS, UPON. The three operators correspond to different temporal relationships that must hold between the time t in which `A` is derived, and the time t' in which $\mathcal{A}$ holds. The semantics is the following:

- WHENEVER derives `A` for each instant in `([begin,end],P)` where $\mathcal{A}$ holds (i.e., $t = t'$).
- ASLONGAS derives `A` for each instant t in `([begin,end],P)` such that $\mathcal{A}$ has been "continuously" true for all $t' < t$ in `([begin,end],P)`.
- UPON derives `A` for each instant t in `([begin,end],P)` such that $\mathcal{A}$ has been true in some $t' < t$ in `([begin,end],P)`.

A graphical representation of the semantics of the different temporal operators is given in Figure 1.1. Note that WHENEVER corresponds to classical implication. For instance, a rule can state that in the summer of year 2002, summer-staff can read a document WHENEVER regular-staff can read it. ASLONGAS embodies a classical implication and a temporal operator. ASLONGAS derives `A` until $\mathcal{A}$ becomes false for the first time. For instance, with ASLONGAS one can formulate a rule stating that regular-staff can read a document

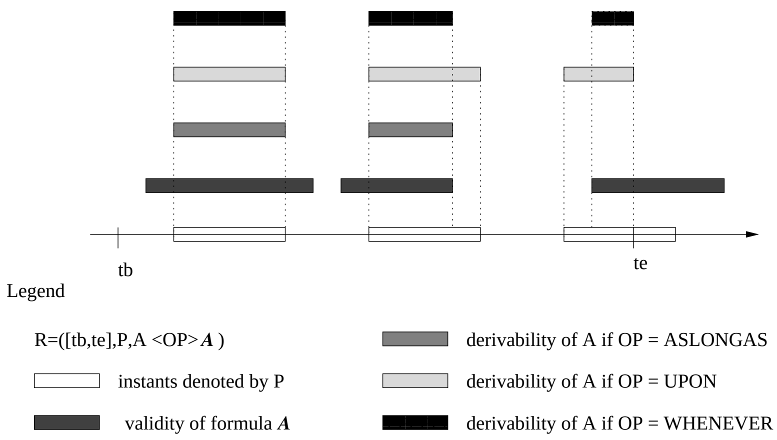


Fig. 1.1. Semantics of the different temporal operators [1.7]

every working day in year 2002, until the first working day in which summer-staff is allowed to read that document. Finally, UPON works like a trigger. For instance, a rule can state that Ann can read pay-checks each working day starting from the first working day in year 2002 in which Tom can write pay-checks.

In this framework, policy specifications are called *Temporal Authorization Bases* (TABs, for short). They are sets of periodic authorizations and derivation rules. TABs are given a semantics by embedding them into *function-free constraint logic programs* over the integers, a fragment of CLP($\mathbf{Z}$) denoted by Datalog$^{\mathbf{not},\equiv\mathbf{Z},<\mathbf{Z}}$.

Temporal expressions are translated into equations and disequations called *periodicity* and *order constraints*. This mathematical formalism is more appropriate for automated deduction as well as for proving formal properties on the specifications. TABs are translated into sets of Datalog$^{\mathbf{not},\equiv Z,<Z}$ clauses as summarized in Table 1.1.[5]

The predicate *valid*() represents the validity of authorizations at specific time points. Predicate $valid^f$ is the analogue of *valid*() for boolean expressions of authorizations. The auxiliary predicates *denied*(), $once\_not\_valid^f()$ and $once\_valid^f()$ are introduced to express quantification. $denied(t, \mathtt{s}, \mathtt{o}, \mathtt{m})$ is true in an interpretation if there is at least one negative authorization $\mathtt{A}$ such that $\mathtt{s(A)} = \mathtt{s}$, $\mathtt{o(A)} = \mathtt{o}$, $\mathtt{m(A)} = \mathtt{m}$, valid at some t. Atom $once\_not\_valid^f(t'', t, \mathtt{P}, \mathcal{A})$ (respectively $once\_valid^f(t'', t, \mathtt{P}, \mathcal{A})$) is true if

[5] In the table, $\mathtt{A}_i$ is used as a shorthand for an authorization 5-tuple. Expressions $\mathtt{A}_i^-$ and $\mathtt{A}_i^+$ force the sign to be negative and positive respectively. $\mathtt{s(A)}$, $\mathtt{o(A)}$, and $\mathtt{m(A)}$ denote the subject, object and operation (or *mode*) fields of the 5-tuple, respectively. The formulas $\mathcal{A}$ occurring after temporal operators are assumed to be in disjunctive normal form. Finally, $t \equiv_{periodicity(\mathtt{P})} y$ means that t mod $periodicity(\mathtt{P})$ equals y mod $periodicity(\mathtt{P})$.

([begin,end], P, A^-) :
$valid(t, \mathtt{A}^-) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$

([begin,end],P, A^+) :
$valid(t, \mathtt{A}^+) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $\mathbf{not}(denied(t,\mathtt{s}(\mathtt{A}^+),\mathtt{o}(\mathtt{A}^+),\mathtt{m}(\mathtt{A}^+))$

([begin,end], P, A^- WHENEVER $\mathcal{A}$) :
$valid(t, \mathtt{A}^-) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $valid^f(t,\mathcal{A})$

([begin,end], P, A^+ WHENEVER $\mathcal{A}$) :
$valid(t, \mathtt{A}^+) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $valid^f(t, \mathcal{A})$, $\mathbf{not}(denied(t,\mathtt{s}(\mathtt{A}^+),\mathtt{o}(\mathtt{A})^+,\mathtt{m}(\mathtt{A}^+))$

([begin,end], P, A^- ASLONGAS $\mathcal{A}$) :
$valid(t, \mathtt{A}^-) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $valid^f(t, \mathcal{A})$, $\mathbf{not}(once\_not\_valid^f(\mathtt{t}_b, t,\mathtt{P},\mathcal{A}))$

([begin,end],P, A^+ ASLONGAS $\mathcal{A}$) :
$valid(t, \mathtt{A}^+) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $valid^f(t, \mathcal{A})$, $\mathbf{not}(once\_not\_valid^f(\mathtt{t}_b, t,\mathtt{P},\mathcal{A}))$,
$\mathbf{not}(denied(t,\mathtt{s}(\mathtt{A}^+),\mathtt{o}(\mathtt{A}^+),\mathtt{m}(\mathtt{A}^+))$

([begin,end], P, A^- UPON $\mathcal{A}$) :
$valid(t, \mathtt{A}^-) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $once\_valid^f(\mathtt{t}_b, t,\mathtt{P},\mathcal{A})$

([begin,end],P, A^+ UPON $\mathcal{A}$) :
$valid(t, \mathtt{A}^+) \leftarrow \mathtt{t}_b \leq t \leq \mathtt{t}_e$, $\textsc{cnstr}(\mathtt{P}, t)$, $once\_valid^f(\mathtt{t}_b, t,\mathtt{P},\mathcal{A})$, $\mathbf{not}(denied(t,\mathtt{s}(\mathtt{A}^+),\mathtt{o}(\mathtt{A}^+),\mathtt{m}(\mathtt{A}^+))$

Auxiliary clauses:

$denied(t, s, o, m) \leftarrow valid(t, s, o, m, -, g)$

$\{\textsc{cnstr}(\mathtt{P}, t) \leftarrow t \equiv_{periodicity(\mathtt{P})} y\}$
$\forall y$ such that $t \equiv_{periodicity(\mathtt{P})} y \Rightarrow t \in \Pi(\mathtt{P})$

$\{once\_valid^f(t'', t,\mathtt{P},\mathcal{A}) \leftarrow t'' \leq t' \leq t,\ \textsc{cnstr}(\mathtt{P}, t'),\ valid^f(t', \mathcal{A})\}$
$\forall$ distinct pair ($\mathtt{P}$,$\mathcal{A}$) appearing in an UPON rule

$\{once\_not\_valid^f(t'', t,\mathtt{P},\mathcal{A}) \leftarrow t'' \leq t' < t,\ \textsc{cnstr}(\mathtt{P}, t'),\mathbf{not}(valid^f(t', \mathcal{A}))\}$
$\forall$ distinct pair ($\mathtt{P}$,$\mathcal{A}$) appearing in an ASLONGAS rule

$\{valid^f(t, \mathcal{A}) \leftarrow \mathbf{not}(valid(t, \mathtt{A}_1)), \ldots, \mathbf{not}(valid(t, \mathtt{A}_k)), valid(t, \mathtt{A}_{k+1}), \ldots, valid(t, \mathtt{A}_m)\}$
$\forall$ distinct conjunction $\mathtt{C} = \bigwedge_{j=1}^{k} \neg\mathtt{A}_j \wedge \bigwedge_{l=k+1}^{m} \mathtt{A}_l$ in $\mathcal{A}$, $k \in [0, m]$, $m \in \mathbf{Z}^+$,
and $\forall$ distinct $\mathcal{A}$ appearing in a derivation rule

Table 1.1. Semantics of periodic authorizations and rules [1.7]

there is at least one instant t' such that $t'' \leq t' < t$, t' satisfies $\mathtt{P}$, and $\mathcal{A}$ is not valid (respectively valid).

The semantics of negation-as-failure is the stable model semantics, extended to constraint logic programs. In order to ensure the uniqueness of the canonical model and its PTIME computability, TABs are restricted in such a way that the corresponding logic program is locally stratified.

In order to implement TAB-based access control efficiently, the canonical model of the corresponding logic program is materialized, that is, it is computed in advance. In this way, access control involves no deduction and is reduced to retrieval. The technical difficulty to be solved is that the canonical model is infinite because time is unbounded. The results of [1.7] show that policy extensions always become periodic after an initial stabilization phase, therefore only this phase and one period need to be materialized. The materialized view is computed using the Dred [1.40] and Stdel [1.57] approaches.

The TABs framework embodies a fixed strategy for conflict resolution (denials-take-precedence). The problem of specifying different strategies is not addressed in [1.7]. Conflict resolution in general will be dealt with in Section 1.2.2.

It is interesting to observe that inference in Datalog$^{\mathbf{not},\equiv\mathbf{Z},<\mathbf{Z}}$ is in general undecidable, whereas the syntactic restrictions satisfied by the embedding's image guarantee decidability. Adopting an ad-hoc policy language is often a good way of enforcing complex syntactic restrictions on the underlying logic transparently, with no extra burden on the users.

Active rules An intermediate approach between imperative and declarative dynamic policy specifications can be found in [1.9]. The specification language is based on active rules called *role triggers*, whose head specifies actions that modify the policy extension. One difference between this language and previous approaches is that dynamic changes concern *roles*, rather than individual authorizations. Mathematically, a role can be regarded as a relation between users and permissions [1.74], so by activating and deactivating roles, active rules simultaneously handle entire groups of authorizations. Moreover, roles are relatively static, as they typically correspond to the organization's structure. Rules can only enable and disable them, role creation and deletion are not supported. This feature makes dynamic policies decidable, even if the sets of users and objects are unbounded, because rules operate only on a finite and fixed set of roles.

The syntax of role activation/deactivation policies is based on *event expressions* and *status expressions*. The former may have the form `enable` R or `disable` R, where R is a role name. Event expressions can be *prioritized* by labeling them—as in p : `enable` R—with a priority p taken from a partially ordered set. If the policy simultaneously entails two conflicting prioritized events p_1 : `enable` R and p_2 : `disable` R, then the event with higher priority overrides the other. If $p_1 = p_2$, then the default choice is p_2 : `disable` R. This choice can be regarded as a particular instantiation of the denial-takes-precedence principle. Status expressions may have the form `enabled` R or $\neg$`enabled` R. Role triggers have the form

$$S_1, \ldots, S_n, E_1, \ldots, E_m \to p : E_0 \texttt{ after } \Delta t$$

where $S_1, \ldots, S_n$ are status expressions, $E_0, \ldots, E_m$ are event expressions ($n, m \geq 0$) and Δt specifies a delay after which E_0 will be executed. Conceptually, all role triggers whose bodies are satisfied fire in parallel and schedule the event in their head. The bodies can be made true by previously scheduled events, by events requested at runtime by the security administrator, and by *periodic events*, that is, prioritized events labeled with a periodic expression of the same form as those adopted in [1.7] (and illustrated previously).

The dynamic behavior of the policies is modeled via a *transition function*, obtained by adapting the stable model semantics to role triggers and periodic events. A suitable form of *stratifiability* is introduced to make the system behavior deterministic and computable in polynomial time. While standard stratifiability takes into account only the dependencies between the head and the bodies of program rules (role triggers, in this case), the new form of stratifiability must take into account also the priorities associated with rule heads, and the temporal delays Δt. Events can be blocked by other simultaneous conflicting events, so there exist a number of implicit negative dependencies between them.

Role triggers are more difficult to use than the temporal constraints of TABs (because of side effects), but the former can be naturally implemented through the standard triggers supported by several DBMS (a prototype implementation based on Oracle is described in [1.9]). Periodic events are materialized like TABs, by considering only the stabilization phase and one period. The materialization, called *agenda*, is then used to generate events that activate triggers, that are in one-to-one correspondence with the active rules of the policy language. In this way, the policy language gives an abstract and cleaner view of the underlying procedural mechanism supported by the DBMS. As a particular example, the syntactic restrictions introduced in the paper make the effects of the actions independent from the order in which triggers are fired.

1.2.2 Hierarchies, inheritance and exceptions

Since the earliest time, computer security models have supported some forms of abstraction on the authorization elements, in order to formulate security policies in a concise fashion. For instance, users can be collected in groups, and objects and operations in classes. The authorizations granted to a user group apply to all its member user and authorizations granted on a class apply on all its members. This is a way of representing concisely sets of authorizations, through an implicit form of qualified universal quantification.

Typically, groups (resp. classes) need not be disjoint and can be nested. The only constraint is that no group (class) should be a member of itself.

Roles can be structured in a similar way, although role hierarchies reflect specialization rather than set inclusion [1.74]. This common structure can be abstracted by the notion of *hierarchy* [1.45]. Hierarchies are triples $(\mathsf{X}, \mathsf{Y}, \leq)$ where:

1. X and Y are disjoint sets
2. $\leq$ is a partial order on $(\mathsf{X} \cup \mathsf{Y})$ such that each $x \in \mathsf{X}$ is a minimal element of $(\mathsf{X} \cup \mathsf{Y})$; an element $x \in \mathsf{X}$ is said to be minimal iff there are no elements below it in the hierarchy, that is iff $\forall y \in (\mathsf{X} \cup \mathsf{Y}) : y \leq x \Rightarrow y = x$.

Here, X may be thought of as the set of "primitive" entities, while Y contains the "aggregate" entities (or "generalized" entities, for roles).

The hierarchies of authorization elements—called *basic hierarchies* in the following—naturally induce a hierarchy of authorizations. For example, if authorizations are simply triples (*subject,action,object*), then let $(s, o, a) \leq (s', o', a')$ iff $s \leq s'$, $a \leq a'$ and $o \leq o'$. In this case, we say that the authorization (s, o, a) is *more specific* than (s', o', a'). Now, if (s', o', a') is in the policy extension, then all the (s, o, a) such that $(s, o, a) \leq (s', o', a')$ are implicitly in the extension. By analogy with object-oriented languages, we say that (s, o, a) is *inherited* from (s', o', a'). This is perhaps the simplest possible form of derivation that can be found in policy languages.

The authorization hierarchy can be exploited to formulate policies in a top-down, incremental fashion. An initial set of general authorizations can be subsequently and progressively refined with more specific authorizations that introduce *exceptions* to the general rules. A related benefit is that policies may be expressed in a very concise and manageable fashion.

Exceptions make inheritance a *defeasible* inference, in the sense that inherited authorizations can be retracted (or *overridden*) as exceptions are introduced. As a consequence, the underlying logic becomes non monotonic.

Exceptions require richer authorizations. It must be possible to say explicitly whether a given permission is granted or denied. Then authorizations are typically extended with a *sign*, '+' for granted permissions and '$-$' for denials.

It may easily happen that two conflicting authorizations are inherited from two incomparable authorizations. Conflicting inheritance may arise even if the basic hierarchies are trees. For example, if $s' \leq s$ and $o' \leq o$, then from the two incomparable authorizations $(s', o, a, +)$ and $(s, o', a, -)$ the two conflicting authorizations $(s', o', a, +)$ and $(s', o', a, -)$ are inherited. The reader may easily verify that the authorization hierarchy can be a tree only when at most one of the basic hierarchies is nontrivial.

Therefore, a policy specification language featuring inheritance and exceptions must necessarily deal with conflicts. One of the simplest popular conflict resolution methods—called *denial-takes-precedence*—consists in overriding the positive authorization with the negative one (i.e., in case of conflicts, the authorization is denied), but this is not the only possible approach.

Recent proposals have worked toward languages and models able to express, in a single framework, different inheritance mechanisms and conflict resolution policies. Logic-based approaches have been investigated as a means to achieve such flexibility.

Flexible Authorization Framework Jajodia et al. [1.45] worked on a proposal for a logic-based language that attempted to balance flexibility and expressiveness on the one side, and easy management and performance on the other. The language allows the representation of different policies and protection requirements, while at the same time providing understandable specifications, clear semantics, and bearable data complexity. Their proposal for a Flexible Authorization Framework (FAF) corresponds to a polynomial (quadratic) time data complexity fragment of default logic.

In FAF policies are divided into four decision stages, corresponding to the following policy components (Figure 1.3).

– *Authorization Table*. This is the set of explicitly specified authorizations.
– The *Propagation policy* specifies how to obtain new derived authorizations from the explicit authorization table. Typically, derived authorizations are obtained according to hierarchy-based derivation. However, derivation rules are not restricted to this particular form of derivation.
– The *Conflict resolution policy* describes how possible conflicts between the (explicit and/or derived) authorizations should be solved. Possible conflict resolution policies include: *no-conflict* (conflicts are considered errors), *denials-take-precedence* (negative authorizations prevail over positive ones), *permissions-take-precedence* (positive authorizations prevail over negative ones), *nothing-takes-precedence* (the conflict remains unsolved). Some forms of conflict resolutions can be expressed within the propagation policy, as in the case of overriding (also known as *most-specific-takes precedence*).
– A *Decision policy* defines the response that should be returned to each access request. In case of conflicts or gaps (i.e., some access is neither authorized nor denied) the decision policy determines the answer. In many systems, decisions assume either the open or the closed form (by default, access is granted or denied, respectively).

Starting from this separation, the Authorization Specification Language of FAF takes the following approach:

– The authorization table is viewed as a database.
– Policies are expressed by a restricted class of stratified and function-free normal logic programs called *authorization specifications*.
– The semantics of authorization specifications is the stable model semantics [1.34]. The structure of authorization specifications guarantees stratification, and hence stable model uniqueness and PTIME computability.

The four decision stages correspond to the following predicates. (Below s, o, and a denote a subject, object, and action term, respectively, where a term is either a constant value in the corresponding domain or a variable ranging over it).

`cando`(o,s,±a) represents authorizations explicitly inserted by the security administrator. They represent the accesses that the administrator wishes to allow or deny (depending on the sign associated with the action).

`dercando`(o,s,±a) represents authorizations derived by the system using logic program rules.

`do`(o,s,±a) handles both conflict resolution and the final decision.

Moreover, a predicate `done` keeps track of the history of accesses (for example, this can be useful to implement a Chinese Wall policy), and a predicate `error` can be used to express integrity constraints. In addition, the language has a set of predicates for representing hierarchical relationships (`hie`-predicates), and additional application-specific predicates, called `rel`-predicates. Hierarchical predicates represent hierarchical relationships within the different components of the system (objects, subjects, or access modes). For instance, in most realistic systems, data items are organized hierarchically. For example, in a file system, the basic objects are files, but these files are typically organized in a hierarchical directory structure. Similarly, in an object oriented database, the objects being accessed are organized into an object hierarchy. In an analogous way, authorization subjects can be basic users or hierarchical groups in which they are organized. Application specific predicates capture the possible different relationships, existing between the elements of the data system, that may need to be taken into account by the access control system. Examples of `rel`-predicates are `owner(user, object)`, which models ownership of objects by users, or `supervisor(user1, user2)`, which models responsibilities and control within the organizational structure.

Authorization specifications are stated as logic rules defined over the above predicates. To ensure stratifiability, the format of the rules is restricted as illustrated in Figure 1.2. Note that the adopted strata reflect the logical ordering of the four decision stages.

The authors of [1.45] present a materialization technique for producing, storing and updating the stable model of the policy. The model is computed on the initial specifications and updated with incremental maintenance strategies.

Note that the clean identification and separation of the four decision stages can be regarded as a basis for a policy specification methodology. In this sense, the choice of a precise ontology and other syntactic restrictions (such as those illustrated in Figure 1.2) may assist security managers in formulating their policies.

Hierarchical temporal authorization model A general approach to authorization inheritance under the denial-takes-precedence principle can be found in [1.8]. In this framework, called *hierarchical temporal authorization*

Stratum	Predicate	Rules defining predicate
0	`hie`-predicates `rel`-predicates `done`	base relations. base relations. base relation.
1	`cando`	body may contain `done`, `hie-` and `rel-`literals.
2	`dercando`	body may contain `cando`, `dercando`, `done`, `hie-`, and `rel-` literals. Occurrences of `dercando` literals must be positive.
3	`do`	in the case when head is of the form $\texttt{do}(\_, \_, +a)$ body may contain `cando`, `dercando`, `done`, `hie-` and `rel-` literals.
4	`do`	in the case when head is of the form $\texttt{do}(o, s, -a)$ body contains just one literal $\neg\texttt{do}(o, s, +a)$.
5	`error`	body may contain `do`, `cando`, `dercando`, `done`, `hie-`, and `rel-` literals.

Fig. 1.2. Rule composition and stratification of the proposal in [1.45]

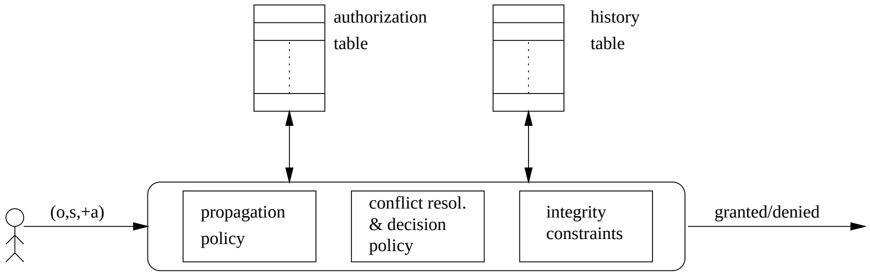


Fig. 1.3. Functional authorization architecture in [1.45]

model (HTAM), no distinction is made between primitive and derived authorizations. This feature required an extension to the classical stratification techniques.

The syntax of the policy language is the same as the syntax of TABs [1.7] with one important difference: the elements of authorization triples can be arbitrary nodes of the basic hierarchies. The authorization hierarchy is defined by: $(s, o, a, sgn, g) \leq (s', o', a', sgn, g)$ iff $s \leq s'$, $a \leq a'$ and $o \leq o'$.

An authorization (s, o, a, sgn, g) (with $sgn \in \{+, -\}$) can be overridden by any more specific authorization $(s', o', a', sgn, g) \leq (s, o, a, sgn, g)$ with a *non-defeasible proof*, that is, a proof that does not rely on inheritance. In case of conflicts between two inherited authorizations, the contradiction is resolved according to the denial-takes-precedence principle. The formal semantics is formulated by adapting the fixpoint construction underlying the stable model semantics.

The major technical difficulty to be solved in this framework is that policy specifications are always equivalent to a non-stratifiable logic program. In general, such programs do not have a unique canonical model (and may have no canonical model at all), and inference is not tractable.

Non-stratifiable programs are programs with recursive calls through negation. In the case of authorization inheritance, given a positive authorization A^+ with a parent A_p^+, and given the corresponding negative authorization A^- with parents $A_1^-, \ldots, A_n^-$, the logic program rules that define inheritance would have a structure similar to:

$$\begin{array}{ll} A^- \leftarrow A_i^-,\ \mathsf{not}\ A^+ & (1 \leq i \leq n) \\ A^+ \leftarrow A_p^+,\ \mathsf{not}\ A^-,\ \mathsf{not}\ A_1^-, \ldots,\ \mathsf{not}\ A_n^-. & \end{array}$$

The first rule says that A^- can be inherited if some of its parents A_i^- is derivable and the conflicting authorization A^+ is not derivable. The second rule says that A^+ can be inherited if some of its parents is derivable, the conflicting authorization A^- is not derivable, and none of A^-'s parents is derivable, that is, A^- *cannot be inherited*. In this way the denial-takes-precedence principle is enforced: A^+ can be inherited only if A^- cannot.

No program containing the above rules is stratifiable, because the two authorizations A^+ and A^- "call" each other through negation-as-failure. This is caused by the negative conditions $\mathsf{not}\ A^+$ and $\mathsf{not}\ A^-$ in the bodies of the two rules, that cannot be removed because they are needed to block inheritance when the opposite authorization has a non-defeasible proof (as in the case of explicit exceptions).

Since the above inheritance rules are implicit in the semantics of the language (i.e., roughly speaking, they belong to every policy even if they do not occur explicitly in the policy), it turns out that the uniqueness of the canonical model of the policy and its PTIME computability cannot be proved by means of the usual stratification techniques. Indeed, [1.8] extends the theory of logic programming by identifying a class of non-stratifiable programs—called *almost-stratifiable programs*—with the same nice properties as stratifiable programs.

Note that the bodies of the above rules are mutually inconsistent, because they contain A_i^- and $\mathsf{not}\ A_i^-$, respectively. If the inheritance rules are the only sources of non-stratifiable cycles, then, roughly speaking, such cycles are "harmless" because the rules that yield such cycles cannot be simultaneously applicable. Intuitively, after computing all the ancestors of A^+ and A^-, one of the above rules can be discarded (because its body is false) and the remaining rules at the same level become stratifiable. Accordingly, the paper introduces a formal definition of a *dynamic* form of stratification, interleaved with the computation of the canonical model. The formal results of [1.8] show that if the policy satisfies a weakened stratification condition (ensuring that all non stratifiable cycles are caused only by the inheritance rules), then the policy has one canonical model computable in polynomial time.

Note that the denial-takes-precedence principle is extremely important for these results. It disambiguates the meaning of the specifications and ensures that the bodies of the inheritance rules involved in a negative cycle are always mutually inconsistent.

HTAM and FAF enjoy complementary properties. On one hand, HTAM gives a general solution to inheritance and overriding, by resorting to non-stratifiable programs. In FAF it is impossible to override an inherited authorization with a derived authorization, because of the syntactic constraints enforcing stratifiability.

On the other hand, the conflict resolution and decision policies are fixed in HTAM (and based on the denials-take-precedence principle, that is necessary for stable model uniqueness and tractability), while FAF supports multiple such policies. Indeed, the main goal of FAF is flexibility. So far, no attempt has been made at combining the advantages of both models.

Ordered logic programs A significantly different approach, inspired by *ordered logic programs* [1.50], can be found in [1.10]. There, security policies generalize the structure of an access control matrix, by introducing inheritance over the matrix indexes, and by allowing derivation rules in the matrix elements. The logic language is inspired by *ordered logic programs* [1.50].

More precisely, let a *reference* be a pair $(object, subject)$, and let references be structured by the natural hierarchy induced by the basic object and subject hierarchies. A rule, in this framework, is a pair

$$\langle\, (o,s),\ L_0 \leftarrow L_1, \ldots, L_m, \mathsf{not}\ L_{m+1}, \ldots, \mathsf{not}\ L_n \,\rangle\,,$$

where (o,s) is a reference. Each L_i is either a standard literal (A or $\neg A$, where A is a logical atom) or a *referential literal* $(o',s').L$, where (o',s') is a reference and L is a standard literal. The authorization predicate has the form $\mathtt{auth}(p,g)$ where p is a privilege (the analogue of the *action* field of the authorizations discussed previously), and g is the grantor of the authorization. As in the previous approaches, the semantics is obtained by adapting the stable model semantics.

This syntax is just a factorized reformulation of the syntax of the other approaches. By default, subject and objects are specified by the rule's reference. In rule bodies, one may refer to other subjects and objects by means of referential literals. The real difference between this approach on one hand, and HTAM and FAF on the other hand, is that when a policy specification has multiple stable models, the authors of [1.10] propose three different conflict resolution strategies:

1. Use the *well-founded* model of the policy. This (partial) model approximates the intersection of all the stable models of the policy, and can be computed in polynomial time.
2. Use the intersection of the stable models (called the *skeptical semantics* of the policy). Computing the intersection is a coNP-hard problem.

3. Select dynamically a stable model that contains all the authorizations granted so far and grants the current operation, if possible. Otherwise deny the operation. The problem of finding such a stable model (called *credulous semantics*) is NP-complete (data complexity). Moreover, the history of previous authorization must be stored and maintained.

The second and third strategies are computationally demanding. There exist powerful engines for computing the skeptical and credulous stable semantics [1.65, 1.32], but so far they have not been experimentally evaluated in this context. A further difficulty related to the third strategy is that the policy cannot be materialized in advance, because its extension is selected dynamically at access control time.

1.2.3 Message control

Many modern systems are based on distributed objects or agents that interact and cooperate by exchanging messages. A natural approach in such systems is to formulate policies at the level of the communication middleware. Messages may be delivered, blocked or modified to enforce the security policy. For example, when the sender is not trusted, the receiver specified in the message may be replaced by a secure wrapper. The message contents may be changed, too—say—by weakening a service request.

This approach is pursued in a series of papers by Minsky et al. (e.g., [1.61, 1.62]). In the former paper, the policy language of the *Darwin* system is described. It adopts a Prolog-like syntax to formulate message handling and transformation rules.

The act of sending a message is denoted by the logical atom $\mathtt{send}(s, m, t)$, where s is the sender object, m is the message, and t is the target object. Note that from a mathematical viewpoint, messages have the same structure as authorization triples (subject,action,object).

Policies consist of sets of *laws*. Laws are functions that map each message $\mathtt{send}(s, m, t)$ onto an action of the form $\mathtt{deliver}(m', t')$ or `fail`. It may be the case that $t \neq t'$ (the message is redirected to another object) or $m \neq m'$ (the message contents are modified).

Each law can be composed of several rules, that are interpreted according to the procedural semantics of Prolog. The syntax is inspired by Definite Clause Grammars (the symbol '`-->`' is equivalent to '`:-`'). Consider the following example:

```
r1: send(S, ^M, T) -->
        isa(T, module) &
        T.owner=S      &
        deliver(^M,T).

r2: send(S, @M, T) -->
        isa(S,module)  &
```

```
        isa(T,module)  &
        deliver(@M,T).
```

The first rule prescribes that every object `S` can send a meta-message `^M` (such as `^new`, to create objects, or `^kill` to destroy objects) to any subclass `T` of the class `module`, provided that `S` is the owner of `T`. The second rule allows arbitrary messages between the system's modules (note that `^` and `@` specify the message type). In these rules, the message and the target are not modified.

The implementation follows two approaches. In the first approach, called *dynamic*, messages are intercepted and transformed by interpreting the policy. The second approach, called *static*, is more efficient. By means of static analysis, program modules are checked to see whether the policy will be obeyed at runtime. When the policy prescribes message modification, the code may have to be changed. Of course, the static approach is applicable only to local modules, under the control of the security administrator.

The second paper [1.62] adapts these ideas to the framework of electronic commerce. Changes mainly concern the set of primitive operations, rule structure is preserved. Moreover, the language distinguishes the act of sending a message from the actual message delivery.

The level of abstraction and the expressiveness of these policy languages are appealing. Unfortunately, semantics is described procedurally, by relying on the user's understanding of Prolog interpreters. No equivalent declarative formulation is provided, even if it seems possible to give a declarative reading to law rules—e.g., in abductive terms.

Another interesting option is applying a policy description language based on event-condition-action rules, such as $\mathcal{PDL}$ [1.55, 1.25], to message handling policies. However, so far $\mathcal{PDL}$ has been considered only in the framework of network management, and static analysis techniques have not been considered as an implementation technique.

1.2.4 Policy composition frameworks

The sources of multiple requirements mentioned at the beginning of this section motivate also the need for *combining* different security policies, developed independently by different departments, organizations or institutions. Similar needs arise when legacy databases have to be integrated into a new information system.

Constraint-based approaches The earliest approaches to policy composition in the literature are based on constraint languages that express relationships between the elements of different policies. For example, in [1.37] a constraint may state that object u can/cannot access object v (in this simplified model subjects and objects coincide), with the restriction that u and v must belong to different policies. This approach, however, is not based on logic, and focuses mainly on computational complexity results.

Another composition framework [1.16] deals with *multilevel* databases, and in particular with the problem of merging different, partially ordered sets of *security levels*. In the multilevel approach, each user and each data object is labeled with a security level belonging to a finite set, partially ordered by a relation $\preceq$. To preserve secrecy, a user with security level ℓ is allowed to read only the objects with label $\ell' \preceq \ell$, and write only those with label $\ell'' \succeq \ell$. In [1.16], different sets of security levels are merged by formulating a set of so-called *interoperation constraints* of the form $\ell_1 : h_1 \preceq \ell_2 : h_2$ or $\ell_1 : h_1 \not\preceq \ell_2 : h_2$ (the latter are called *negative constraints*), where h_i is one of the given partially ordered sets to be merged and ℓ_i is a security level in h_i $(i = 1, 2)$. The intended meaning of these constraints is that in the merged ordering, the security level ℓ_1 of h_1 must be (resp., must not be) below the security level ℓ_2 of h_2. The orderings h_i occur explicitly in the constraints in order to disambiguate the cases in which the same name has been used with different meanings in different orderings. Conversely, two security levels with different names can be identified with a symmetric pair of constraints, such as $\ell_1 : h_1 \preceq \ell_2 : h_2$ and $\ell_2 : h_2 \preceq \ell_1 : h_1$.

The result of merging the given security levels is formalized following two guiding principles:

1. The interoperation constraints should be satisfied by the composition (trivial as it may seem, this requirement is not formulated explicitly in [1.37], where some negative constraints may be violated by the composition).
2. The original orderings should not be modified by the composition, that is, for all levels ℓ and ℓ' of any given ordering h, ℓ is below ℓ' in the merged ordering if and only if ℓ is below ℓ' in h. In [1.37], the equivalent of this requirement is formulated as a pair of properties: the *principle of autonomy* (the permissions granted by an individual source are granted also by the composition) and the *principle of security* (the composition does not introduce any new permissions within any individual source).

The result of merging the given orderings, called a *witness of composability*, consists of a new ordering m and a family of *translation functions* $\varphi_i : h_i \to m$ that map the original security levels onto the new levels of m. If the witness satisfies all the interoperation constraints and preserves the original orderings, then the given orderings are *composable* (w.r.t. the given constraints).

The results of [1.16] show that: (i) if the orderings are composable, then the witness is unique up to isomorphism, and (ii) composability can be checked in quadratic time (the algorithm effectively constructs a witness).

If the sources are not composable with respect to the given interoperation constraints IC, then [1.37, 1.16] consider the problem of relaxing the constraints to some $IC' \subset IC$ that allows composability. In [1.37] it is shown that finding such an IC' with maximal cardinality is an NP-complete problem, and this result is extended to the refined framework in [1.16]. On the contrary, [1.16] shows that the relaxation problem is in PTIME if IC' is only required to be maximal w.r.t. set inclusion and—possibly—w.r.t. some preference relation over the constraints of IC.

In [1.16] logic plays two roles. First of all, the constraint language is in fact a logical language. Second, the merging problem is axiomatized with a set of Horn clauses. This is a neat and convenient way of showing some uniqueness and complexity results, by applying the theory of logic programs. Horn clauses can also be used to implement the composability check, but the complexity of this approach is not optimal—it becomes cubic due to the order transitivity axiom that has three variables. The quadratic complexity bound is obtained with a graph-based algorithm.

A simplified version of the constraint system of [1.16] can be found in [1.31]. This works is not focused only on composition and deals with query folding in a mediated framework.

The problem of merging heterogeneous security orderings—and its solution—can be easily adapted to the problem of merging heterogeneous user and object hierarchies (cf. Section 1.2.2). An example can be found in [1.8].

Deontic approach Deontic logics are modal logics, where the modal operators O, P and F mean 'obligatory', 'permitted' and 'forbidden', respectively.

Cuppens, Cholvy, Saurel and Carrère [1.30] tackle the problem of merging security policies expressed in a deontic logic. They focus on conflict detection and resolution.

The given policies may adopt different vocabularies, that can be related to each other by means of first-order formulas. Further axioms formalize the properties of the domain of discourse. These axioms constitute the so-called *domain knowledge* that—in the examples provided in [1.30]—are essentially Horn clauses with equality.

Technically, when different policies contain conflicting specifications, inconsistencies are avoided by indexing the deontic operators with the name of the rules that occur in the given policies. In [1.30], a rule r is a definite Horn clause whose head is a deontic literal of the form $OP_r\alpha$, where OP_r is an indexed deontic operator (O_r, P_r or F_r) and α is a logical atom. Indexing suffices to distinguish the context in which a norm has been formulated. By means of indexed operators, *conflicting rules* can be axiomatized. Two rules r_1 and r_2 are conflicting if there exists a suitable first-order formula s (describing what the authors call a *situation*), such that the union of the domain knowledge with the set of policy rules entails one of the following formulas:

1. $s \to F_{r_1}\alpha \wedge P_{r_2}\alpha$ (*normative contradiction*)
2. $s \to F_{r_1}\alpha \wedge O_{r_2}\alpha$ (*moral dilemma*).

Suppose s holds. In the first case, α is forbidden by r_1 and permitted by r_2. In the second case, α is forbidden by r_1 and obligatory according to r_2.

Since $O_{r_2}\alpha \to P_{r_2}\alpha$ is an axiom of the deontic logic adopted by [1.30], moral dilemmas turn out to be a special case of normative contradictions. Moreover, the formula in point 1 is equivalent to $(P_{r_1}\alpha \vee F_{r_2}\alpha) \to \neg s$, so the problem of finding normative conflicts can be reduced to inferring $\neg s$ (where s is a situation) from formulas of the form $(P_{r_1}\alpha \vee F_{r_2}\alpha)$. An inference rule called SOL-deduction can be applied to generate such s from the given sets of rules, after translating the modal language into first-order clausal logic. The important technical property of SOL-deduction is that it can be focused on consequences s that belong to a given sublanguage (in this case, the set of sentences that describe a situation).

When normative conflicts have been identified, they can be solved by applying a strategy formulated in a suitable meta-language. The meta-language has:

- predicates for describing rules, policies and their components as terms,
- predicates for stating that a given policy is more specific than another policy,[6]
- predicates for expressing preferences on policy rules.

For example a strategy $S1$ stating that more specific rules are preferred (and hence override) less specific rules can be formulated as follows:

$$\begin{aligned}
&\forall r_1, \forall r_2, preferred(S1, r_1, r_2) \leftrightarrow \\
&\qquad rule(r_1) \wedge rule(r_2) \wedge \\
&\qquad \exists reg_1, \exists reg_2, \\
&\qquad\quad regulation(reg_1) \wedge regulation(reg_2) \wedge \\
&\qquad\quad more_specific(reg_1, reg_2) \wedge \\
&\qquad\quad belongs_to(r_1, reg_1) \wedge belongs_to(r_2, reg_2)\,.
\end{aligned}$$

The composition constraint language and the strategy language are very expressive. The only limitation is that non monotonic inferences are not tackled. In building a system based on this approach, a correct and complete formulation of the domain knowledge is critical for security verification and conflict detection. From the implementation viewpoint, optimization and scalability issues have not been investigated.

Algebraic approach A more general approach to policy composition is taken in [1.21], where no assumption is made on the languages used to specify the given policies. The authors note that non monotonic policy specification languages can combine different policies, but the result is a *monolithic* specification, which is hard to maintain and verify. The paper provides a list of desiderata for a better policy composition framework:

1. *Heterogeneous policy support.* The composition framework should be able to combine policies expressed in arbitrary languages and enforced by

[6] The authors show how to recognize specificity automatically, in some cases. When the automated technique cannot be applied, specificity is declared explicitly, much like a preference relation.

different mechanisms. For instance, a data warehouse may collect data from different data sources whose security restrictions may be stated with different specification languages, or refer to different paradigms (e.g., open vs closed policy).

2. *Support of unknown policies.* It should be possible to leave part of the policies unknown. It should also be possible to import policies specified and enforced in external systems. These policies are like "black-boxes" for which no (complete) specification is provided. They can be queried at access control time. Think, for example, of a situation where given accesses are subject to a policy P enforcing "central administration approval". While P can respond "yes" or "no" to each specific request, neither the description of P, nor the complete set of accesses that it allows might be available. Run-time evaluation is therefore the only possible option for P. In the context of a more complex and complete policy including P as a component, the specification could be partially compiled, postponing at runtime only the evaluation of P (and its possible consequences).
3. *Controlled interference.* Policies cannot always be combined by simply merging their specifications, even if they are formulated in the same language. This could have undesired side effects so that the combined policy might not reflect the specifications correctly. As a simple example, consider the combination of two systems P_{closed}, which applies a closed policy, of the form "s can apply operation a to object o if a positive authorization $(s, o, +a)$ is in the policy", and P_{open} which applies an open policy, of the form "s can apply a to o if no negative authorization $(s, o, -a)$ is in the policy". In the union of the two specifications, the latter decision rule would derive all the authorizations not blocked by P_{open}, regardless of the contents of P_{closed}. Similar problems may arise from uncontrolled interaction of the derivation rules of the two specifications. Moreover, if the adopted language is a logic language with negation, then the merged program might not be stratified (which may lead to ambiguous or undefined semantics).
4. *Expressiveness.* The language should be able to express conveniently and in a uniform language a wide range of policy combinations (spanning from minimum privileges to maximum privileges, encompassing priority levels, overriding, confinement, refinement, etc.). The different kinds of combinations must be expressed without changing the input specifications and without ad-hoc extensions to authorizations (like those introduced by some authors to support priorities). For instance, consider a department policy P_1 regulating access to documents and the central administration policy P_2. Assume that access to administrative documents can be granted only if authorized by both P_1 *and* P_2. This requisite can be expressed in prior approaches only by extending explicitly all the rules dealing with administrative documents with the additional conditions specified by P_2. One of the drawbacks of this approach is the resulting

rule explosion and the complex structure and loss of control over the two specifications which, in particular, cannot be maintained and managed autonomously anymore.

5. *Support of different abstraction levels.* The composition language should help the identification of the policy components and their interplay at different levels of abstraction. This feature is important to: *i)* facilitate specification analysis and design; *ii)* facilitate cooperative administration and agreement on global policies; *iii)* support incremental specification by refinement.
6. *Formal semantics.* The composition language should be declarative, implementation independent, and based on a solid formal framework. An underlying formal framework is needed to: *i)* ensure non-ambiguous behavior and *ii)* reason about policy specifications and prove properties on them [1.48].

These issues are simultaneously tackled by introducing a suitable *policy algebra.* To make the algebra compatible with a large number of policy specification languages, the algebra is designed to operate only on policy *extensions*, independently from how they have been specified. The only assumption is that three sets S, O, and A denoting the subjects, objects, and actions, respectively, are given, and authorizations are the triples in $\mathsf{S}\times\mathsf{O}\times\mathsf{A}$.[7] Depending on the application context and the policy to be enforced, subjects could be users or groups thereof, as well as roles or applications; objects could be files, relations, XML documents, classes, and so on.

Authorization terms are triples of the form (s, o, a), where s is a constant in S or a variable over S, o is a constant in O or a variable over O, and a is a constant in A or a variable over A. A *policy* is a set of ground (positive) authorization terms (that constitute the extension of the policy).

Intuitively, a policy represents the outcome of an authorization specification, where, for composition purposes, it is irrelevant how specifications have been stated and their outcome computed.

The algebra (among other operations) allows policies to be restricted (by posing constraints on their authorizations) and closed under given sets of inference rules. To make the algebra compatible with many languages for constraining authorizations and formulating rules, the algebra is parametric w.r.t. the following languages and their semantics (i.e., different languages can be plugged in):

1. An *authorization constraint language* $\mathcal{L}_{acon}$ and a semantic relation $\mathsf{satisfy} \subseteq (\mathsf{S}\times\mathsf{O}\times\mathsf{A})\times\mathcal{L}_{acon}$; the latter specifies for each ground autho-

[7] Policies containing more complex authorizations including signs and grantors can be modelled with a set of simpler policies based on triples in $\mathsf{S}\times\mathsf{O}\times\mathsf{A}$ (e.g., by introducing one triple-based policy for each (sign,grantor) pair, cf. [1.21]).

rization term (s, o, a) and constraint $c \in \mathcal{L}_{acon}$ whether (s, o, a) satisfies c.[8]

2. A *rule language* $\mathcal{L}_{rule}$ and a semantic function $\mathsf{closure} : \wp(\mathcal{L}_{rule}) \times \wp(\mathsf{S} \times \mathsf{O} \times \mathsf{A}) \rightarrow \wp(\mathsf{S} \times \mathsf{O} \times \mathsf{A})$; the latter specifies for each set of rules R and ground authorizations P which authorizations are derived from P by R.

The syntax of algebraic *policy expressions* is given by the following grammar:

$$E ::= \mathbf{id} \mid E + E \mid E \& E \mid E - E \mid E\,\hat{}\,C \mid o(E,E,E) \mid E * R \mid T(E) \mid (E)$$
$$T ::= \tau\, \mathbf{id}.T \mid \tau\, \mathbf{id}.E$$

Here **id** is the token type of policy identifiers, E is the nonterminal describing *policy expressions*, T is a construct called *template*, that represents incomplete policy expressions, C and R are the constructs describing $\mathcal{L}_{acon}$ and $\mathcal{L}_{rule}$, respectively (they are not specified here because the algebra is parametric w.r.t. $\mathcal{L}_{acon}$ and $\mathcal{L}_{rule}$).

The semantics of algebraic expressions is a function that maps each expression onto a set of ground authorizations, that is, a policy. The simplest possible expressions, namely identifiers, are bound to sets of triples by *environments*.

An environment e is a partial mapping from policy identifiers to sets of ground authorizations. By $e[X/S]$ we denote a modification of environment e such that

$$e[X/S](Y) = \begin{cases} S & \text{if } Y = X \\ e(Y) & \text{otherwise} \end{cases}$$

In symbols, the semantics of an identifier X w.r.t. an environment e will be denoted by $[\![X]\!]_e \stackrel{\text{def}}{=} e(X)$.

The meaning of the policy composition operators is described below. in the following, let the meta-variables P and P_i range over policy expressions:

Addition (+) merges two policies by returning their union. Formally, let $[\![P_1 + P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \cup [\![P_2]\!]_e$. Addition yields the *maximum privilege* of P_1 and P_2, that is, an authorization is granted if at least one of the two policies grants it.

Conjunction (&) merges two policies by returning their intersection. Formally, $[\![P_1 \& P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \cap [\![P_2]\!]_e$. Intuitively, conjunction corresponds to the *minimum privilege* granted by the two policies.

Subtraction (-) The formal semantics is $[\![P_1 - P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \setminus [\![P_2]\!]_e$. Intuitively, the second argument of subtraction specifies exceptions to the authorizations of the first argument. Subtraction encompasses the functionality of *negative authorizations*. The advantages of subtraction over explicit denials include a simplification of the conflict resolution policies

[8] Constraints involving more than one tuple are handled via the rule language illustrated in item 2.

and a clearer semantics, as discussed in [1.21]. Subtraction can also be used to express different overriding/conflict resolution criteria as needed in each specific context, without affecting the form of the authorizations.

Closure ($*$) closes a policy under a set of inference rules. The general definition is $[\![P * R]\!]_e \stackrel{\text{def}}{=} \mathsf{closure}(R, [\![P]\!]_e)$. Derivation rules can, for example, enforce propagation of authorizations along hierarchies (inheritance), or enforce more general forms of implication, related to the presence or absence of other authorizations.

Scoping restriction ($\hat{}$) restricts the application of a policy to a given set of subjects, objects, and actions. Formally, $[\![P\hat{}c]\!]_e \stackrel{\text{def}}{=} \{(s,o,a) \mid (s,o,a) \in [\![P]\!]_e,\ (s,o,a)\ \mathsf{satisfy}\ c\}$, where $c \in \mathcal{L}_{acon}$. Scoping is particularly useful to "limit" the statements that can be established by a policy and to enforce authority confinement. Intuitively, all authorizations in the policy which do not satisfy the scoping restriction are ignored.

Overriding (o) replaces part of a policy with a corresponding fragment of a second policy. The portion to be replaced is specified by means of a third policy. Formally, $[\![o(P_1, P_2, P_3)]\!]_e \stackrel{\text{def}}{=} [\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$.

Template (τ) defines a partially specified policy that can be completed by supplying the parameters. $[\![\tau X.P]\!]_e$ is a function over policies (ground authorization sets), such that for all policies S, $[\![\tau X.P]\!]_e(S) \stackrel{\text{def}}{=} [\![P]\!]_{e[X/S]}$. Templates can be instantiated by applying them to a policy expression. For all policy expressions P_1, $[\![(\tau X.P)(P_1)]\!]_e \stackrel{\text{def}}{=} [\![\tau X.P]\!]_e([\![P_1]\!]_e) = [\![P]\!]_{e[X/[\![P_1]\!]_e]}$. We say that all the occurrences of X in an expression $\tau X.P$ are *bound*. The *free* identifiers of a policy expression P are all the identifiers with non-bound occurrences in P. Clearly, $[\![P]\!]_e$ is defined iff all the free identifiers in P are defined in e.

Templates are useful for representing partially specified policies, where some component X is to be specified at a later stage. For instance, X might be the result of further policy refinement, or it might be specified by a different authority. When a specification P_1 for X is available, the corresponding global policy can be simply expressed as $(\tau X.P)(P_1)$. Templates with multiple parameters can be expressed and applied using the following abbreviations:

$$\tau X_1, \ldots, X_n.P = \tau X_1.\tau X_2.\ldots.\tau X_n.P$$
$$(\tau X_1, \ldots, X_n.P)(P_1, \ldots, P_n) = (\ldots((\tau X_1, \ldots, X_n.P)(P_1))(P_2)\ldots)(P_n)\,.$$

Figure 1.4 summarizes the policy composition operators and their semantics. A nice feature of the algebra is that policy specifications can be formulated in a *graphical language*. Figure 1.4 illustrates two possible graphical representations for each operator. Another advantage of an algebraic language is that it is relatively familiar for the users of relational database languages. Indeed, the fragment of the algebra without closure and templates

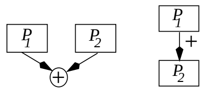
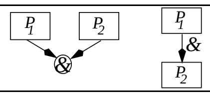
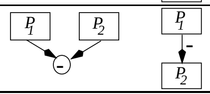
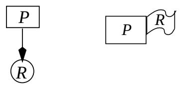
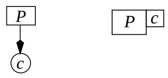
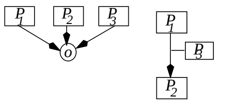
Operator	Semantics $[\![\ ]\!]_e$	Graphical representation
$P_1 + P_2$	$[\![P_1]\!]_e \cup [\![P_2]\!]_e$	
$P_1 \& P_2$	$[\![P_1]\!]_e \cap [\![P_2]\!]_e$	
$P_1 - P_2$	$[\![P_1]\!]_e \setminus [\![P_2]\!]_e$	
$P * R$	$\mathsf{closure}(R, [\![P]\!]_e)$	
$P\hat{\ } c$	$\{t \in [\![P]\!]_e \mid t \text{ satisfy c}\}$	
$o(P_1,P_2,P_3)$	$[\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$	

Fig. 1.4. Operators of the algebra and their graphical representation

is a relational algebra over relations with a fixed schema ([1.21] gives a formal account of this claim).

To illustrate the use of the algebra let us consider a simple example.

Consider a policy regulating access to university laboratories. To use machines, students must be authorized by both the lab tutors and the department administration. The tutors and the administration can specify authorizations at different levels of details. In particular, policy P_{tutors}, specified by the tutors, can state permissions on a single-user/single-machine basis, with statements like `(jim.smith,machine1, login)`. The department can state permissions with reference to groups of students and machines, with statements like `(cs101,cs-lab,login)` which, closed under classical propagation rules implies a permission for all students enrolled in class `cs101` to use machines in the `cs-lab`. Authorized accesses are defined as a conjunction of the two policies (intuitively, students should have permission to use the laboratory by the department and machine assignment by the tutor). In addition, access to any laboratory resource is forbidden to students blacklisted for infraction to rules (e.g., honor code); only an explicit permission by the provost can override such a restriction. The overall policy can thus be expressed as

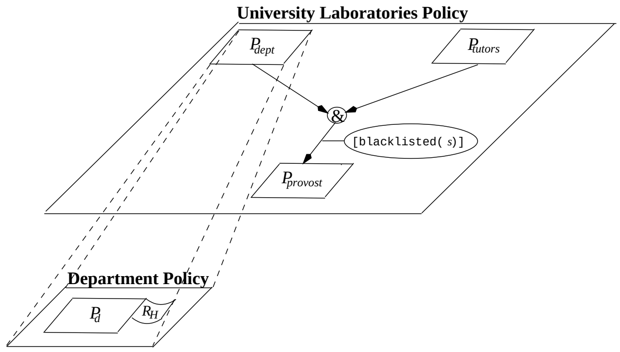


Fig. 1.5. An example of graphical representation of policy algebra

$o(P_{tutors} \& P_{dept}, P_{provost}, \hat{}[\texttt{blacklisted}(s)])$ whose graphical representation is illustrated in Figure 1.5.

In [1.21] it is shown how to translate algebraic specifications into normal logic programs. The reason is that one can then apply well-known *partial evaluation techniques* to materialize compound policies. Intuitively, all the sub-policies that are given a complete specification are reduced to sets of ground authorizations (i.e., their extensions is computed) while the incomplete parts of the specification are simplified to an optimized program with suitable calls to the modules that implement the policy (see [1.21] for further details).

The expressiveness of the algebra is discussed extensively in [1.21], by reproducing examples from the literature and by evaluating the algebra w.r.t. the list of desiderata. In particular,

1. *Heterogeneous policies* can be supported either by exploiting the algebra constructs to represent the different policies or by referring to heterogeneous policies through policy identifiers then interpreted by means of wrappers.
2. *Unknown policies* are supported by means of policy identifiers that can remain unbound in the environment. The translation of expressions into logic programs and the application of partial evaluation techniques on them allow us to delay the evaluation of unknown policies until run time, without need of redoing the whole computation, and guarantee the correctness of the resulting controls. Furthermore, templates allow the expression of incomplete policies in the formal semantics and the proofs of correctness properties on incomplete specifications.
3. *Interference* of program rules and authorizations coming from different policies is controlled by restricting rule application to specific policies by means of the closure construct.

4. *Expressiveness* is achieved by the different operators that easily allow the formulation of protection restrictions as illustrated in the examples and discussions contained herein.
5. *Different abstraction levels* are naturally supported by the component-based approach. Each component may internally be structured in sub-components and security administrators can zoom in the different policies or look at a higher level view as desired. Templates provide a formal means to operate on the different levels as one may simply look at the template (higher abstraction level) or at its actual parameters corresponding to the contents (zoom in) of the formal parameters.
6. *Formal semantics* on which the algebra is based semantics can be exploited to reason about properties of the specifications and not only to implement them. The algebra is implementation independent and can be used to design, analyze, and combine requirements in different systems.

1.2.5 Credential-based access control

The clean separation between authentication and authorization made by traditional access control models is limiting for modern open systems. Today, many activities are based on the use of remote resources and services, and on the interaction between different, remotely located parties that may know little about each other. Service requests may be submitted by remote, previously unknown parties, not just by the local community of users. Therefore, servers cannot be assumed to be able to identify and authenticate each of these possible parties.

This paradigm shift is witnessed by a large body of work moving toward alternative approaches for establishing identities or properties of requesters (e.g., [1.14, 1.17, 1.53, 1.84]). These proposals adopt digital certificates—or *credentials*—as a means to establish properties of their holder, such as identity, accreditation, or authorizations. The properties stated by credentials are certified by given entities, such as certification authorities.

A second consequence of the open nature of this scenario is that requestors cannot be assumed to know which certificates should be submitted together with their request. The required credentials may have to be described by the server by sending a message back to the requestor. In turn, the requestor may want to release sensitive data (such as credit card numbers) only to "certified" servers. For this purpose, the requestor may ask the server for credentials, too. The overall interaction process between the two parties may then evolve toward a form of negotiation [1.62, 1.69] oriented to trust management. Then security policies (and specification languages) have to deal with both what should be communicated between the parties, and how to communicate. Particular care must be taken to avoid improper disclosure of information during negotiation.

Logic-based approaches have been adopted in this area for specifying, reasoning about, and communicating protection requirements. Current investigations include: certificate-based authorizations enforcement and delegation (e.g., [1.26, 1.52, 1.78]), trust management (e.g., [1.14, 1.53, 1.84]), negotiation [1.69] and message exchange [1.62].

The first line of work investigating the use of logic for expressing credential-based authorization is by Winslett's group [1.46, 1.71, 1.78]. The approach in [1.78] is motivated by the following observations: *i)* it is not reasonable to expect a client to submit all its credentials regardless of their relevance to the current request, *ii)* the client's pool of credentials may be large and the server may ask for arbitrary subsets of this pool; *iii)* different servers may trust different authorities, and may ask for certificates concerning entirely different properties. Based on these observations, the authors of [1.78] propose a programming methodology for writing credential acceptance policies that can be downloaded from the server to the client. The methodology uses logic programming because of the expressiveness and portability of the resulting specifications, as well as because of its natural fit in enforcing credential recognition and field extraction. On the server's side, a set of Prolog rules defines the credential acceptance policy (CAP) that associates each request with the required credentials. In response to a request, the server replies with the credential acceptance policy applicable to the request. Then the client determines the necessary credentials by evaluating such policy at its side. This initial proposal has been further extended by considering different strategies for protecting the parties' privacy, for selecting credentials to enable a request, and for trust management (e.g., [1.84, 1.53]).

In [1.78], the vocabulary is not restricted, and during negotiation policy rules are communicated as they are. Bonatti and Samarati [1.17] refine this approach by proposing a specialized logic-based policy language and a filtering mechanisms for protecting policy contents. The language can be used for two purposes:

– Establishing protection requirements on the servers.
– Restricting credentials and declarations release. This applies to both servers and clients.

In the remainder we illustrate this approach, that explicitly concerns the use of logic for authorization specification and enforcement.

The proposal in [1.17] is based on the concept of *portfolio* which denotes the properties a party can submit in order to obtain (or offer) services. These properties can be *data declarations* (e.g., identity, address, and credit card number) and *credentials* (i.e., digital certificates). The difference between them is that declarations are non-certified statements uttered by the party (e.g., name, address, and credit card number), while credentials are uttered and signed (i.e., certified) by authorities trusted for making the statements. The functionalities offered by a server are defined by a set of services. As usual, declarations, certificates, and services are classified into suitable hierarchies, for the purposes recalled in Section 1.2.2.

The language adopts the following predicates. Their intuitive meaning is expressed with respect to a *state* that maintains the parties' profiles and the status of the current negotiation, including the current request, the certificates already received, and so on.

- A binary predicate symbol, credential, whose first argument is a credential term and whose second argument is a public key term. Intuitively, a ground atom $\mathsf{credential}(c, K)$ is true iff the current state contains a credential c verifiable with public key K.
- A predicate symbol, declaration, whose argument is an element of the form "*attribute_name* = *value_term*". Intuitively, a ground atom $\mathsf{declaration}(d)$ is true iff the current state contains declaration d.
- A binary predicate symbol cert_authority, whose arguments are a certification authority and its public key. Intuitively, $\mathsf{cert\_authority}(CA, K_{CA})$ states that the party trusts certificates signed by certification authority CA whose public key is K_{CA}.
- A set of non predefined *state* inquiry predicates that query information stored locally (*persistent state*) or acquired during a negotiation (*negotiation state*). As a special case, state predicates may be built-ins and calls to external packages [1.82]. They can be used to query information that is not available at the site (e.g., via profiles) but must be retrieved through certain procedures.
- A set of non predefined *abbreviation* predicates, used to express concisely complex requests for declarations, credentials, and chains thereof. Intuitively, abbreviation define "macros" that can be used as a shorthand for boolean combinations of requirements.
- A set of standard built-in mathematic predicates, including $=, \neq, <$.

With the above predicates, servers can specify the rules regulating access to their services (*service rule*), and both parties can formulate rules for regulating credential and declaration release (*release rules*). Service and release rules are logical rules whose body is composed from basic predicates. Rule heads specify which services are accessible, or which portfolio objects are releasable, according to the rule.

As an example, the two service rules

$\mathsf{service\_reqs}(\mathsf{buy}()) \leftarrow \mathsf{declaration}(\text{credit\_card\_number}{=}X).$

$\mathsf{service\_reqs}(\mathsf{buy}()) \leftarrow \mathsf{current\_customer}(U), \mathsf{customer\_affiliation}(I),$
$\quad \mathsf{credential}(\mathsf{authorized\_to\_buy}(\text{issuer}{=}I, \text{user}{=}U, \text{user\_key}{=}K_U), K_I),$
$\quad \mathsf{cert\_authority}(I, K_I).$

dictate that service buy can be granted to clients that submit either a declaration stating their credit card number or a credential authorizing them for the purchase signed by the organization they are affiliated with. Here, current_customer and customer_affiliation are state predicates.

An example of release rule is the following:

$\mathsf{release\_reqs}(\text{credit\_card\_info}) \leftarrow \mathsf{declaration}(\text{no\_disclosure}{=}\texttt{"accept"}),$
$\quad \mathsf{current\_service}(\mathsf{buy}()) .$

which dictates that credit card information (which is an abstraction including the credit card number as well) can be released only in the context of buying requests (state information) and upon reception of a non disclosure agreement from the counterpart.

To illustrate how the rule evaluation works, for simplicity, we summarize the intended meaning of the rules and the filtering process from the point of view of the server (the client's point of view is symmetrical).

The naive way of requesting a set of declarations and credentials—namely, listing all the possible sets that would enable the service—is not feasible, due to the large number of possible alternatives. This can be avoided by giving the client a compact requirement description, based on abbreviation predicates. More precisely, the server sends a set of rules R (that define the relevant abbreviation predicates), and the client searches its portfolio for a set of credentials C such that $R \cup C$ entails the authorization for the service (the readers with a logical background have probably noted the similarity between credential search and abduction[9]).

However, the server cannot simply send the rules R to the client. Some of these rules may query the server's private state information (e.g., to check whether the client is a `good_customer`). Clearly, the server should not send its private information to the client; it should then evaluate state predicates at its side. State predicate evaluation must be done *before* sending the rules to the client, because any set of credentials matching the server's request should suffice to access the desired service. In principle, certain parts of the state, such as the client's preferences, need not be hidden, but they should be evaluated anyway before sending the rules to the client. The reason is that the client is not expected to be asked again and again for the same profile information submitted during previous interactions (profiles are maintained by the server precisely for the purpose of making client-server interactions concise and less redundant).

On the client's side symmetrical issues arise. The client may ask the server for declarations or credentials before releasing parts of its portfolio, and such requests must be expressed in a compact way. Again, this is accomplished by sending a set of rules (that define the abbreviation predicates) to the server, after simplifying away the state predicates.

The *rule filtering* process described so far is formalized by exploiting the *partial evaluation* techniques developed for logic programs, and the associated semantics preservation result. They guarantee that the original set of rules R is equivalent (in a suitable sense) to the set of filtered rules R', so that the other party can equivalently use R' for selecting the set of credentials C. The filtering process includes also the selection of the rules relevant to the

[9] We recall that abduction is a reasoning process that given a set of logical formulae T, a set of *abducible* formulae A, and a proposition Ψ, finds out a set of abducibles $A' \subseteq A$ such that $T \cup A'$ entails Ψ. The underlying logic may be classical, modal, or non monotonic.

current request, and possibly a *renaming phase* to transform rule heads into anonymous abbreviation atoms.

The paper ([1.17]) briefly mentions an implementation based on XSB [1.72], a Prolog system supporting goal tabulation and delay. This choice is motivated by the following properties of XSB: *i)* It combines the advantages of top-down derivations (that are focused on the goal to be proved and can deal with function symbols and infinite domains) and bottom-up derivations (that can be computed in polynomial time). *ii)* The tabling mechanism of XSB caches the results of previous derivations. If used appropriately, this can be very useful as one derivation can be used several times within a single session. *iii)* Some of the built-ins of XSB—originally designed for dealing with non-stratified negation—can be easily adapted to collect the sets of required credentials/declarations.

In general, the set of credentials C fulfilling the request is not unique. The problem of selecting the set of credential according to some optimality criterion (e.g., based on the sensitivity of the information encoded in the credential) is not tackled in [1.17].

1.3 Policy evaluation and verification

The preceding sections show how powerful and sophisticated policy specification languages can be. Clearly, as policies become more complex, even declarative and modular specification languages cannot prevent security administrators from inserting errors in their policies. Therefore, some automated or semi-automated policy verification tools are needed.

Here we are not concerned with the problem of whether a particular system's behavior correctly reflects a given policy (point 3 of Section 1.1). We are rather concerned with the problem of identifying potential errors in the policy formulation. In this respect, examples of relevant questions are:

- Is a given policy (or policy module) empty (i.e., does the policy grant no authorization at all)? More generally, we may ask whether each object can be accessed by at least one subject. If this is not the case, then that object cannot be used. These kind of checks has to do with the system's *availability* and is largely independent from the application.
- Similarly, we may ask whether the policy contains all possible authorizations (or, equivalently, no access is ever denied).
- Application-dependent checks may need to decide whether a particular authorization can possibly be granted by the policy, at some time. For example, in order to obey privacy laws, we may ask whether any sensitive piece of data can ever be accessed without the owner's explicit authorization.

Of course, we are interested in static verification tools. Unfortunately, as we already pointed out in Section 1.2.1, the extension of the policy is sometimes undecidable.

For example, Harrison, Ruzzo and Ullman [1.42] show that it cannot be decided whether a given authorization can become part of a dynamic policy (this is known as the *safety problem*). They prove their undecidability result for a simple imperative policy language supporting user and object creation and deletion.

The other temporal policy languages illustrated in Section 1.2.1 are decidable, and so is the safety problem. However, there is an independent problem affecting both these languages and static languages. The problem is that policy specifications may be partially unknown or unavailable at verification time. For example, the precise set of users and objects may be unknown in advance. Moreover, some policy modules may be either specified later, or be always unavailable as a whole—e.g., because they are formulated by a different organization. Then *the desired properties should hold for all possible ways in which the missing details can be filled in.*

If policies were cast into a monotonic logic, incomplete policies could be naturally handled by standard inference. However, the logic underlying policy languages is typically non monotonic, and in such formalisms, the lack of information may support new conclusions—say, through negation-as-failure. An appropriate verification procedure must be able to distinguish whether an authorization is missing because it will not be included in the actual policy at runtime, or whether the authorization is missing simply because that part of the policy is not known in detail.

This kind of non monotonic reasoning had not been tackled before by the literature on non monotonic logics. It has been introduced in [1.18] for the purpose of verifying complex security policies and agent programs. Since all the languages mentioned so far can be embedded into logic programs based on the stable model semantics or variants thereof, the problem of verifying the policies written in these languages can be given a general formulation and general solutions by tackling it in the framework of logic programs.

The notion of incomplete specification is formalized by *open logic programs* [1.18]. They are triples $\langle P, F, O \rangle$ where P is a normal logic program (the incomplete specification), F is a set of function and constant symbols that do not occur in P, and O is a set of predicate symbols called *open predicates* (that may occur in P). Intuitively, F and O provide the syntactic material that can be used to complete the partial specification P. The set F provides the symbols that can be used to extend the program's domain. For example, F may contain an infinite supply of login names and object identifiers. The set O identifies the predicates that are not completely defined by the rules in P.

The complete specifications that can be obtained from $\Omega = \langle P, F, O \rangle$, called the *completions* of Ω, are all the normal logic programs P' such that

1. $P' \supseteq P$,
2. the constant and function symbols of P' occur in P or F,
3. if $r \in P' \setminus P$, then $head(r) \in O$.

In other words, the rules that occur in P' and not in the incomplete specification P must be constructed from the vocabulary of P extended with F and O, and must define one of the open predicates.

The set of all the completions of $\langle P, F, O \rangle$ is denoted by $\mathsf{Comp}(P, F, O)$. There exist four kinds of *open inference*. They are derived from the two basic forms of inference supported by the stable model semantics: a sentence Ψ is a *credulous* consequence of a logic program P if Ψ holds in *some* stable model of P, while Ψ is a *skeptical* consequence of a logic program P if Ψ holds in *all* the stable models of P. In the following, by "consistent program" we mean a program with at least one stable model (all normal logic programs are classically consistent, but some have no stable models).

1. (Credulous open inference) $\langle P, F, O \rangle \models^c \Psi$ iff for some $P' \in \mathsf{Comp}(P, F, O)$, P' credulously entails Ψ.
2. (Skeptical open inference) $\langle P, F, O \rangle \models^s \Psi$ iff for all $P' \in \mathsf{Comp}(P, F, O)$, P' skeptically entails Ψ.
3. (Mixed open inference I) $\langle P, F, O \rangle \models^{cs} \Psi$ iff for some consistent $P' \in \mathsf{Comp}(P, F, O)$, P' skeptically entails Ψ.
4. (Mixed open inference II) $\langle P, F, O \rangle \models^{sc} \Psi$ iff for each consistent $P' \in \mathsf{Comp}(P, F, O)$, P' credulously entails Ψ.

Intuitively, credulous open inference and mixed inference of type I can be used to check whether the complete policy can possibly have an undesirable property Ψ, when the underlying application semantics is credulous or skeptical, respectively. In the simplest case, Ψ can simply be an undesirable authorization.

Similarly, skeptical open inference and mixed inference of type II can be used to check whether the complete policy will necessarily have a desirable property Ψ, when the underlying application semantics is credulous or skeptical, respectively.

The consistency conditions in mixed open inference are needed for technical reasons. Most open programs $\langle P, F, O \rangle$ have some inconsistent completion, so, without any consistency requirement, mixed open inference I would entail every sentence Ψ from such $\langle P, F, O \rangle$, while mixed open inference II would entail no Ψ. In other words, the two entailment relations would be trivial in most cases.

The four open entailments are pairwise dual.

Proposition 1.3.1. *(Duality) For all open programs Ω and all sentences Ψ,*

1. $\Omega \models^c \Psi$ *iff* $\Omega \not\models^s \neg\Psi$;
2. $\Omega \models^{sc} \Psi$ *iff* $\Omega \not\models^{cs} \neg\Psi$.

As a consequence, in principle, it suffices to implement two of the four inference relations.

In order to understand the relative strength of the four entailments, note that they form a diamond-shaped lattice.

Proposition 1.3.2. *(Entailment lattice) Suppose there exists a consistent $P' \in \mathsf{Comp}(\Omega)$. Then, for all sentences Ψ,*

1. *$\Omega \models^{s} \Psi$ implies $\Omega \models^{cs} \Psi$ and $\Omega \models^{sc} \Psi$;*
2. *$\Omega \models^{cs} \Psi$ implies $\Omega \models^{c} \Psi$;*
3. *$\Omega \models^{sc} \Psi$ implies $\Omega \models^{c} \Psi$.*

The problem of computing open inference is still under investigation. In [1.18] two extensions of resolution are introduced for skeptical open inference and mixed inference of type I, respectively. The goals are conjunctions of implications, and there are 5, 6 inference rules, respectively. The so-called *failure rule* handles negation-as-failure. It is restricted to predicates $p \notin O$, as expected. The failure rule is expressed in terms of a *counter-support* function that abstracts the actual mechanism for computing negation-as-failure. From the implementation viewpoint, this is the most delicate part of the calculus. In the general setting, a complete implementation of the counter-support function is impossible. Fortunately, the translation of the policy composition algebra into logic programs (cf. Section 1.2) enjoys syntactic restrictions that greatly simplify counter-support computation.

Besides the resolution calculi, in some restricted cases one can apply standard engines such as *DLV* [1.32] and *Smodels* [1.65], thanks to the relationships between open programs and answer set programming investigated in [1.19]. It may also be possible to apply abductive procedures, thanks to the relationships investigated in the same paper. Currently, the applicability of DLV and Smodels to such tasks is limited by their being unable to handle infinite domains. So when the sets of users, objects and operations are not all finite, DLV and Smodels can only perform approximate checks on domains with bounded size.

1.4 Other applications of logic to security

Policy specification and verification are only some of the applications of logics in computer security. The description of other applications lies beyond the scope of this chapter, however we shall briefly mention some of the most interesting areas related to the innovative applications of databases.

Trust management Modern distributed applications must solve the problem of whether remote users and remotely generated agents can be trusted when they claim to have certain properties. Logics are playing an interesting role in this area, both as tools for specifying and investigating the meaning of signed statements (e.g., electronic credentials) [1.23, 1.81, 1.2, 1.4, 1.44, 1.51, 1.41, 1.27], and as flexible policy specification languages (cf. Section 1.2.5).

Model checking for protocol verification There is a vast body of literature on automated theorem proving for model checking and process verification. Some successful approaches use logic programming techniques [1.67]. Security protocols are sufficiently complex to hide subtle bugs, but sufficiently simple to be analyzed with automated verification tools derived from formal methods. Logic programming techniques are being applied in this area. As an incomplete sample of recent works we mention [1.60, 1.63, 1.56, 1.24, 1.1]. Of course, security protocols are intimately related with security in distributed and open systems.

Intrusion detection Given that a complete security analysis of real computer systems is not feasible, intrusions cannot simply be prevented. It is necessary to build tools for detecting unexpected security violations. Machine learning and data mining techniques are being applied for this purpose. Some intrusion detection systems produce *classification rules* (for identifying violations), that are similar to logic program rules, that is, these system embody some form of logic program *induction*. A non-exhaustive list of recent works in this direction is [1.33, 1.54, 1.49].

1.5 Conclusions and perspectives

In this chapter we have illustrated the applications of different logics to the specification and enforcement of security policies. We surveyed different proposals where logic-based formulations helped making progress in the formalization of security problems and their solution.

It is interesting to notice that the cross-fertilization between computer security and logics has been fruitful for both areas.

As for the field of computer science logics, the areas of research on automated deduction and computational logics found new motivations in security applications. The investigation of specific application cases for which no theoretical solutions existed originated new studies, especially in the field of logic programming and non monotonic reasoning. For example, the theory of normal logic programs has been extended with new interesting classes of programs, such as almost-stratified programs (Section 1.2.2). The theory of non monotonic reasoning has been extended to cover non monotonic inferences from incomplete theories (Section 1.3).

As for the security field, the advantages have been illustrated in the course of the chapter. The formal foundations of logic-based policy languages naturally support formal verification techniques and tools at different levels of abstraction, as required by the sensitive nature of secure applications. Logic-based policy languages provide the flexibility and expressiveness needed for specifying, composing, and enforcing sophisticated security policies. It is worth noticing that the positive features of logic-based solutions and their advantages over competing approaches have apparently not yet had a proportional impact on real applications and practical scenarios. A possible explanation is that the intended users of policy languages are reluctant to get into formal details, and prefer easy to use (yet much less powerful) approaches. Then a careful balance between expressive power and manageability may improve the diffusion of logic-based approaches in real systems. Policy languages can be simplified by replacing first-order syntax with a different, user-oriented syntax. For example, several of the languages illustrated in this survey have no logical variables, even if these languages can be embedded into suitable first-order logic programming languages. The implicit use of logical variables is common in knowledge representation [1.80]. As an additional benefit, the ad-hoc language sometimes singles out a decidable and tractable fragment of a generally more complex logic.

There are still several areas in which logic-based approaches may contribute to the state of the art in security research.

So far, most of the efforts have been focused on one aspect of security called *confidentiality*, that deals with the protection of confidential information from unauthorized disclosure. Another important aspect is *integrity*, that deals with data corruption. This aspect is partly addressed by means of integrity constraints in databases, and partly by means of authorizations for restricting update operations. However, integrity is not only concerned with data access, as data may also be corrupted due to hardware failures, blackouts, malicious employees, and so on. Such threats affect also an aspect of security called *availability*, concerned with the interruption of system functionality. While countermeasures to such threats cannot be given a software solution, still formal analysis tools could be helpful to single out possible security threats. These aspects are currently not covered by logic-based approaches. It would be interesting to explore formal approaches to security analysis in its most general sense.

Two further aspects of security, related to each other, are *violation detection* and *repair*. The former is tackled by means of machine learning techniques with some analogies to logic program induction (cf. Section 1.4). The latter covers confinement and removal of security violations. Repair-oriented tools might benefit from diagnostic reasoning and abductive reasoning, to isolate the causes of violation. So far, the only support to repair in logic-based policy languages can be found in [1.9], where security managers are allowed to issue role enabling/disabling commands at runtime, possibly overriding the standard policy. Moreover, security managers may introduce restrictions for individual users, to handle emergency situations.

In fact, something similar to abduction and diagnostic reasoning is already adopted in some approaches for policy verification (see the "intentional answers" of [1.29] and credential selection in [1.17]). However, the possible roles of abduction in security are not yet completely understood.

The forms of negotiation that naturally arise in open system security (cf. Section 1.2.5) involve a number of technical issues (including negotiation length and optimization, and the problems related to the interoperation of heterogeneous parties) that are far from being exhaustively investigated. The work developed in the context of credential-based approaches and provisional authorization solutions represents only a starting point for this problem.

References

1.1 M. Abadi, B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 2002), pp. 33-44, ACM, 2002.

1.2 M. Abadi. On SDSI's Linked Local Name Spaces. *Journal of Computer Security* 6(1-2), 1998.

1.3 M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15:706–734, 1993.

1.4 A.W. Appel, E.W. Felten. Proof-Carrying Authentication. 6th ACM Conference on Computer and Communications Security, 52-62, 1999.

1.5 K.R. Apt, H.A.Blair, A. Walker. Towards a theory of declarative knowledge. In J. Minker (ed.), *Foundations of deductive databases and logic programming*, pp 89-148, Morgan Kaufmann, 1988.

1.6 D. Bell, L. LaPadula. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4(2/3), 1996.

1.7 E. Bertino, C. Bettini, E. Ferrari and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS*, 23(3), 1998.

1.8 E.Bertino, P.A.Bonatti, E.Ferrari, M.L.Sapino. Temporal authorization bases: from specification to integration. *Journal of Computer Security*, 8(4), 2000.

1.9 E. Bertino, P.A. Bonatti, E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. on Information and System Security*, 4(3):191-223, 2001.

1.10 E. Bertino, F. Buccafurri, E. Ferrari and P. Rullo. A logical framework for reasoning on data access control policies. *Proc. of the 1999 IEEE Computer Security Foundations Workshop*, 1999.

1.11 P. Bieber, F. Cuppens. A logical view of secure dependencies. *Journal of Computer Security*, 1(1):99-129, 1992.

1.12 J. Biskup. For unknown secrecies refusal is better than lying, *Data and Knowledge Engineering*, 33:1-23, 2000.

1.13 J. Biskup, P.A. Bonatti. Lying versus refusal for known potential secrets. *Data and Knowledge Engineering*, 38(2):199-222, 2001.

1.14 M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Springer Verlag – LNCS State-of-the-Art series, 1998.

1.15 P.A.Bonatti, S.Kraus, V.S.Subrahmanian. Foundations of secure deductive databases. *IEEE Trans. on Knowledge and Data Engineering*, 7:406–422 (1995).

1.16 P.A. Bonatti, M.L. Sapino, V.S. Subrahmanian. Merging heterogeneous security orderings. *Journal of Computer Security*, 1997.

1.17 P. Bonatti, P. Samarati. Regulating Service Access and Information Release on the Web, *Proc. of the Seventh ACM Conference on Computer and Communications Security*, 2000. To appear in the Journal of Computer Security.
1.18 P.A. Bonatti. Reasoning with open logic programs. Proc. of LPNMR'01, pp. 147-159, LNAI 2173, Springer, 2001.
1.19 P.A. Bonatti. Abduction, ASP and Open Logic Programs. Proc. of the International Workshop on Nonmonotonic Reasoning (NMR'02), Toulouse, France, 2002.
1.20 P. Bonatti, E. Damiani, S. De Capitani di Vimercati, PSamarati. An Access Control System for Data Archives, *Proc. of the 16th International Conference on Information Security*, Paris, France, June 2001.
1.21 P.A. Bonatti, S. De Capitani Di Vimercati, P. Samarati. An Algebra for Composing Access Control Policies, *ACM Transactions on Information and System Security*, 5(1):1-35, 2002.
1.22 D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proc. Symp. on Security and Privacy*, pages 215–228, Oakland, CA, 1989.
1.23 M. Burrows, M. Abadi, R. Needham. A logic of authentication. *ACM Transactions on Computer Systems* 8,(1):18-36, 1990.
1.24 L. Carlucci Aiello, F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. on Computational Logic*, 2(4):542-580, 2001.
1.25 J. Chomicki, J. Lobo, S.A. Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. *Proc. of the Seventh International Conference on Principles of Knowledge Representation and Reasoning* (KR 2000), pp. 121-132, Morgan Kaufmann, 2000.
1.26 Y-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, M. Strauss. REFEREE: trust management for Web applications. *World Wide Web Journal*, 2(3):706-734, 1997.
1.27 D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, R.L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security* 9(4):285-322, 2001.
1.28 F. Cuppens. A logical analysis of authorized and prohibited information flow. *Proc of the IEEE Symposium on Security and Privacy*, pp. 100-109, 1993.
1.29 F. Cuppens, C. Saurel. Specifying a security policy: a case study. *Proc. of the Ninth Computer Security Foundations Workshop*, 1996.
1.30 F. Cuppens, L. Cholvy, C. Saurel, J. Carrère. Merging Security Policies: Analysis of a Practical Example. *Proc. of the 11th IEEE Computer Security Foundations Workshop* (CSFW 1998), pp. 123-136, IEEE Computer Society, 1998.
1.31 S. Dawson, X. Qian, P. Samarati. Providing security and interoperation of heterogeneous systems. *Distributed and parallel Databases*, 8(1):119-145, 2000.
1.32 T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In J. Dix et al. (eds.), *Proc. of LPNMR'97*, LNAI 1265, Springer Verlag, 1997
1.33 T. Fawcett, F.J. Provost. Adaptive Fraud Detection. *Data Mining and Knowledge Discovery*, 1(3):291-316, 1997.
1.34 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 1070–1080, MIT Press, 1988.
1.35 J.I. Glasgow, G.H. MacEwen, P. Panangaden. A logic for reasoning about security. TOCS 10(3):226-264, 1992.
1.36 J. Goguen, J. Meseguer. Unwinding and inference control. *Proc of the IEEE Symposium on Security and Privacy*, 1984.

1.37 L. Gong, X. Qian. Computational issues in secure interoperation. *IEEE Trans. on Software Engineering*, 22(1):43-52, 1996.
1.38 L. Gong. *Inside Java 2 platform security*. Addison Wesley, 1999.
1.39 J.W. Gray, P.F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73-90, 1998.
1.40 A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 157–166, Washington, D.C., May 1993.
1.41 J.Y. Halpern, R. van der Meyden. A Logic for SDSI's Linked Local Name Spaces. *Journal of Computer Security* 9(1/2):105-142, 2001.
1.42 M.A. Harrison, W.L. Ruzzo, J.D. Ullman. Protection in operating systems. *Comm. ACM*, 19(8):461-471, 1976.
1.43 M.A. Harrison, W.L. Ruzzo, J.D. Ullman. Monotonic Protection Systems. In R.A. DeMillo et al. (eds.), *Foundations of Secure Computation*, Academic Press, New York, 1978.
1.44 J. Howell, D. Kotz. A Formal Semantics for SPKI. ESORICS 2000, LNCS 1895, 140-158, Springer, 2000.
1.45 S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible supporting for multiple access control policies. *ACM Transactions on Database Systems*, 26(2): 214-260, 2001.
1.46 V. Jones, N. Ching, M. Winslett. Credentials for Privacy and Interoperation. *Proceedings of the New Security Paradigms '95 Workshop*, 1995.
1.47 B.W. Lampson, M. Abadi, M. Burrows, E. Wobber. Authentication in distributed systems: Theory and practice. *TOCS*, 10(4):265-310.
1.48 C. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, 1981.
1.49 W. Lee, S.J. Stolfo, K.W. Mok. A Data Mining Framework for Building Intrusion Detection Models. *Proc. of IEEE Symposium on Security and Privacy*, pp. 120-132, 1999.
1.50 N. Leone, P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621-642, 1993.
1.51 N. Li. Local Names in SPKI/SDSI *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pp. 2-15, 2000.
1.52 N. Li, J. Feigenbaum, B. Grosof. A logic-based knowledge representation for authorization with delegation. *Proc. of the 12th IEEE Computer Security Foundations Workshop*, pp. 162-174, 1999.
1.53 M. Li, W.H. Winsborough, J.C. Mitchell. Distributed Credential Chaim Discovery in Trust Managament. *Proceedings of 8th ACM Computer and Communication Security*, 2001.
1.54 U. Lindqvist, P.A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). *Proc. of IEEE Symposium on Security and Privacy*, pp. 146-161, 1999.
1.55 J. Lobo, R. Bhatia, S.A. Naqvi. A Policy Description Language. *Proc. of the of the Sixteenth National Conference on Artificial Intelligence* (AAAI 99), pp. 291-298, AAAI Press, 1999.
1.56 G. Lowe. Towards a Completeness Result for Model Checking of Security Protocols. Journal of Computer Security 7(1), 1999.
1.57 J. Lu, G. Moerkotte, J. Schu and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, May 1995.
1.58 J. McCarthy. Applications of circumscription in formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

1.59 J. McLean. Security models. In *Encyclopedia of Software Engineering*. Wiley Press, 1994.
1.60 Catherine Meadows. The NRL Protocol Analyzer: An Overview. JLP 26(2): 113-131, 1996.
1.61 N.H. Minsky, D. Rozenshtein. A software development environment for law-governed systems. In P.B. Henderson (ed.), *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (SDE'88), pp. 65-75, 1988.
1.62 N.H. Minsky, V. Ungureanu. A Mechanism for Establishing Policies for Electronic Commerce. In *Proc. of the 18th International Conference on Distributed Computing Systems* (ICDCS 1998), pp. 322-331, IEEE Computer Society, 1998.
1.63 J. C. Mitchell, M. Mitchell, U. Stern, Automated Analysis of Cryptographic Protocols Using Murϕ. 1997 IEEE Symposium on Security and Privacy, 141-153, 1997.
1.64 R.C. Moore. Semantical considerations on non-monotonic logic. *Artificial Intelligence*, 25 (1), (1985).
1.65 I. Niemelä, P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In J. Dix, U. Furbach, A. Nerode (eds.), *Logic Programming and Nonmonotonic Reasoning: 4th international conference, LPNMR'97*, LNAI 1265, Springer Verlag, Berlin, 1997.
1.66 L. Palopoli, C. Zaniolo. Polynomial-Time Computable Stable Models, *Annals of Mathematics and Artificial Intelligence*, 17(3,4):261-290, 1996.
1.67 C.R. Ramakrishnan. A Model Checker for Value-Passing Mu-Calculus Using Logic Programming. *Practical Aspects of Declarative Languages, Third International Symposium* (PADL 2001), LNCS 1990, pp. 1-13, Springer, 2001
1.68 R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81-132, (1980).
1.69 M. Roscheisen, T. Winograd. A Communication Agreement Framework for Access/Action Control. *Proc. of 1996 IEEE Symposium on Security and Privacy*, pp. 154-163, 1996.
1.70 D. Saccà, C. Zaniolo. Partial stable models, stable models and non-determinism in logic programs with negation. *Proc. of ACM-PODS*, 1990.
1.71 M. Winslett, N. Ching, V. Jones, I. Slepchin. Using Digital Credentials on the World-Wide Web. *Journal of Computer Security*, 5(3), 1997.
1.72 K. Sagonas, T. Swift, D.S. Warren, J. Freire, P. Rao. The XSB programmer's manual, Version 2.2. http://xsb.sourceforge.net, 2000
1.73 P. Samarati, S. De Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In R. Focardi and R. Gorrieri (eds.), *Foundations of Security Analysis and Design*, Springer-Verlag, LNCS 2171, 2001.
1.74 R. Sandhu. Role-based Access Control. *Advances in Computers*, vol. 46, Academic Press, 1998.
1.75 R. Sandhu, P. Samarati. Authentication, Access Control and Intrusion Detection. In A. Tucker (ed.), *CRC Handbook of Computer Science and Engineering*, CRC Press Inc., 1997.
1.76 R. Sandhu. Undecidability of Safety for the Schematic Protection Model with Cyclic Creates *Journal of Computer and System Sciences*, 44(1), February 1992.
1.77 R.S. Sandhu. The typed access matrix model. In *Proc. of 1992 IEEE Symposium on Security and Privacy*, pages 122–136, Oakland, CA, May 1992.
1.78 K. E. Seamons, W. Winsborough, M. Winslett. Internet Credential Acceptance Policies. *Proceedings of the Workshop on Logic Programming for Internet Applications*, 1997.

1.79 G.L. Sicherman, W. de Jonge, R.P. van de Riet. Answering queries without revealing secrets, *ACM Transactions on Database Systems*, 8(1):41-59, 1983.
1.80 J.F. Sowa. *Knowledge representation: logical, philosophical, and computational foundations*, Pacific Grove: Brooks/Cole, 2000.
1.81 S.G. Stubblebine, R.N. Wright. An Authentication Logic Supporting Synchronization, Revocation, and Recency. ACM Conference on Computer and Communications Security, 95-105, 1996.
1.82 V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu,A. Rajput, T. Rogers, R. Ross, C. Ward. Hermes: Heterogeneous reasoning and mediator system.
http://www.cs.umd.edu/projects/hermes/publications/abstracts/hermes.html.
1.83 T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107-136, 1993.
1.84 T. Yu, M. Winslett, K.E. Seamons. Interoperable Strategies in Automated trust Negotiation. *Proceedings of 8th ACM Computer and Communication Security*, 2001.