Chapter 1

RULE-BASED POLICY SPECIFICATION

Grigoris Antoniou
Information Systems Laboratory, Institute of Computer Science, FORTH, Greece
antoniou@ics.forth.gr

Matteo Baldoni
Dipartimento di Informatica,Università degli Studi di Torino, Torino, Italy
baldoni@di.unito.it

Piero A. Bonatti
Università di Napoli Federico II, Naples, Italy
bonatti@na.infn.it

Wolfgang Nejdl
L3S Research Center and University of Hannover, Hannover, Germany
nejdl@l3s.de

Daniel Olmedilla
L3S Research Center and University of Hannover, Hannover, Germany
olmedilla@l3s.de

Abstract Policy-based access control is nowadays a common mechanism to protect data in distributed environments. However, the word policy has been given many different meanings and is used in different contexts. This chapter gives an overview of the existing approaches to logic- and rule-based system behavior specification in the light of the peculiar needs of business and security rules.

Keywords: rule, policy, security, trust management, business rule, action language

Introduction

For a long time, logic programming and rule-based reasoning have been proposed as a basis for policy specification languages. However, the term "policy" has not been given a unique meaning. In fact, it is used in the literature in a broad sense that encompasses the following notions:

- *Security Policies* pose constraints on the behaviour of a system. They are typically used to control permissions of users/groups while accessing resources and services.
- *Trust Management policy languages* are used to collect user properties in open environments, where the set of potential users spans over the entire web.
- *Action Languages* are used in reactive policy specification to execute actions like event logging, notifications, etc. Authorizations that involve actions and side effects are sometimes called *provisional*.
- *Business Rules* are "statements about how a business is done" [25] and are used to formalize and automatize business decisions as well as for efficiency reasons. They can be formulated as *reaction rules*, *derivation rules*, and *integrity constraints* [142, 147].

All these kinds of specification interact tightly with each other: Credential-based user properties are typically used to assign access control permissions; logging, monitoring, and other actions are part of the high-level security specification documents of many organizations; many business rules—e.g., for granting discounts or special services—are based on the same kind of user properties that determine access control decision. Moreover, this kind of business decisions and access control decisions are to be taken more or less simultaneously—e.g. immediately before service access.

There has been extensive research focusing on each of these different notions of policies. In the next four sections, we will give an overview of existing approaches for each of these notions (security policies, policy-based trust management, action languages and business rules), and will then discuss in section 5 two different efforts to integrate several aspects in a common framework. There is still a lot of work to be done in the future and this chapter provides the knowledge and pointers required for anyone planning to work on this area.

1. Security Policies

Rule-based languages are commonly regarded as the best approach to formalizing security policies. In fact, most of the systems we use every day adopt

policies formulated as rules. Roughly speaking, the access control lists applied by routers are actually rules of the form: "*if a packet of protocol X goes from hosts Y to hosts Z then [don't] let it pass*". Some systems, like Java, adopt procedural approaches. Access control is enforced by pieces of code scattered around the virtual machine and the application code; still, the designers of Java security felt the need for a method called *implies*, reminiscent of rules, that causes certain authorizations to entail other authorizations [81].

The main advantages of rule-based policy languages can be summarized as follows:

- People (including users with no specific training in computers or logic) spontaneously tend to formulate security policies as rules.
- Rules have precise and relatively simple formal semantics, be it operational (rewrite semantics), denotational (fixpoint-based), or declarative (model theoretic). Formal semantics is an excellent help in implementing and verifying access control mechanisms, as well as validating policies.
- Rule languages can be flexible enough to model in a unified framework the many different policies introduced along the years as ad-hoc mechanisms. Different policies can be harmonized and integrated into a single coherent specification.

In particular, logic programming languages are particularly attractive as policy specification languages. They enjoy the above properties and have efficient inference mechanisms (linear or quadratic time). This property is important as in most systems policies have to manage a large number of users, files, and operations—hence a large number of possible authorizations. And for those applications where linear time is too slow, there exist well-established compilation techniques (materialization, partial evaluation) that may reduce reasoning to pure retrieval at run time.

Another fundamental property of logic programs is that their inference is *nonmonotonic*, due to *negation-as-failure*. Logic programs can make default decisions in the absence of complete specifications. Default decisions arise naturally in real-world security policies. For example, *open* policies prescribe that authorizations by default are granted, whereas *closed* policies prescribe that they should be denied unless stated otherwise. Other nonmonotonic inferences, such as authorization inheritance and overriding, are commonly supported by policy languages.

For all of these reasons, rule languages based on nonmonotonic logics eventually became the most frequent choice in the literature. A popular choice consists of *normal logic programs*, i.e. sets of rules like

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

interpreted with the *stable model semantics* [73]. In general, each program may have one stable model, many stable models, or none at all. There are opposite points of view on this feature.

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 [31]. For instance, the models of a policy may correspond to all possible ways of assigning permissions that preserve a *Chinese Wall* policy [45]. However, the set of alternative models may grow exponentially, and the problem of finding one of them is NP-complete. There are exceptions with polynomial complexity [124, 132], though.

Some authors believe that security managers would not trust the system's automatic choice and adopt restrictions such as *stratifiability* [6] to guarantee that the canonical model be unique. The system rejects non-stratified specifications, highlighting nonstratified rules to help the security administrator in reformulating the specifications. As a further advantage, stratifiability-like restrictions yield PTIME semantics.

A nonmonotonic logic has been proposed for the first time as a policy specification language by Woo and Lam [151]. The main motivations are expressiveness and flexibility. To address complexity issues (inference in propositional 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 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.

The approach by Woo and Lam has been subsequently refined by many authors. 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*, whereas 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 specifying the policy.

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. Further rule-based languages for policy specification that do not exhibit dynamic or nonmonotonic features will be discussed in the section devoted to trust management.

Dynamic Policies

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

Temporal Authorization Bases In [28] the sets of users and objects are fixed, and the temporal validity of authorizations is specified through *periodic expressions* and suitable *temporal operators*.

Periodic authorizations are obtained by labeling each rule with a *temporal expression* specifying the time instants at which the rule applies. Temporal expressions 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` (for example, `[2/2002,8/2002]`). The rule is valid at all times that lie within the interval `[begin,end]` and satisfy the periodic expression `P`.

Rules are expressions $A \langle \text{OP} \rangle \mathcal{B}$, where A is the authorization to be derived, $\mathcal{B}$ 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 $\mathcal{B}$ holds. The semantics is the following:

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

Note that WHENEVER corresponds to classical implication, ASLONGAS embodies a classical implication and a temporal operator, and UPON works like a trigger.

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$^{\mathbf{not},\equiv\mathbf{Z},<\mathbf{Z}}$ [110, 143].

The semantics of negation as failure is the stable model semantics, extended to constraint logic programs. To ensure the uniqueness of the canonical model and its PTIME computability, TABs are restricted so that the corresponding logic program is locally stratified.

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 [28] 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 [86] and Stdel [110] approaches.

The TABs framework embodies a fixed strategy for conflict resolution (denials take precedence). The problem of specifying different strategies is not addressed in [28]. Conflict resolution in general will be dealt with in Sect. 1.0.

Active Rules An intermediate approach between imperative and declarative dynamic policy specifications can be found in [29]. The specification language, TRBAC, 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 [133], so by activating and deactivating roles, active rules simultaneously handle entire groups of authorizations.

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 ¬`enabled` R. Role triggers have the form

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

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 heads. The bodies can be made true by previously scheduled events, by events requested at run time 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 [28] (and illustrated previously).

Semantics 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. The new form of stratifiability must take into account the priorities associated with rule heads and the temporal delays Δt.

Role triggers can be naturally implemented through the standard triggers supported by several DBMS (a prototype implementation based on Oracle is described in [29]). 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 the triggers. TRBAC gives an abstract and cleaner view of the procedural trigger mechanism supported by the DBMS. For example, the semantics of the triggers derived from TRBAC policies does not depend on the order in which triggers are fired.

Other approaches Two recent languages, KAoS and Ponder [144, 57], adopt constructs similar to active rules for expressing policies. They can formulate *obligations*, that cause an agent to execute some actions whenever a specified event triggers the rule and some precondition is satisfied. In KAoS and Ponder, however, triggers and preconditions cannot refer to other authorizations, so, for example, it is impossible to express rules like "*grant/deny A if A' is granted/denied*". Extending KAoS and Ponder with such rules is a nontrivial problem. In particular, KAoS is based on a description logic (OWL) that extended with rules becomes easily undecidable.

Another recent approach, PROTUNE, supports actions in a significantly different way. Some predicates, called *provisional predicates*, can be made true—if desired—by executing suitable actions. The directions on how and when a provisional predicate should be made true are described in a meta-policy. PROTUNE is described later in this chapter.

Hierarchies, Inheritance and Exceptions

Since the earliest time, computer security models have supported some forms of abstraction on the authorization elements, to formulate security policies concisely. For instance, users can be collected in groups, and objects and operations in classes. The authorizations granted to a user group apply to all of its member users, and authorizations concerning a class of objects apply to all of its members. This is modelled via an authorization hierarchy derived from the hierarchies of subjects, objects and operations—called *basic hierarchies* in the following. For example, if authorizations are simply triples (*subject,object,action*), 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 granted by the policy, then all (s, o, a) such that $(s, o, a) \leq (s', o', a')$ are implicitly granted, too. By analogy with object-oriented languages, we say that (s, o, a) is *inherited* from (s', o', a').

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

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

Recent proposals have worked towards languages and models able to express, in a single framework, different inheritance mechanisms and conflict resolution policies. Logic-based approaches, so far, are the most flexible and expressive.

Flexible Authorization Framework Jajodia et al. [97] attempted to balance flexibility and expressiveness on one side, and easy management and performance on the other. Their proposal for a *flexible authorization framework* (FAF) is a fragment of stratified normal programs with polynomial (quadratic) time data complexity. In FAF, policies are divided into four decision stages, corresponding to the following policy components:

- *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. For instance, derived authorizations can be obtained by inheritance and exceptions.
- 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), and *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).

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. 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 Fig. 1.1. Note that the adopted strata reflect the logical ordering of the four decision stages.

The unique stable model of the given policy can be produced, stored and incrementally updated via suitable materialization techniques.

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

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`	When head is of the form `do`$(\_, \_, +a)$ body may contain `cando`, `dercando`, `done`, `hie-` and `rel-` literals.
4	`do`	When head is of the form `do`$(o, s, -a)$ body contains just one literal $\neg$`do`$(o, s, +a)$.
5	`error`	Body may contain `do`, `cando`, `dercando`, `done`, `hie-`, and `rel-` literals.

Figure 1.1. Rule composition and stratification of the proposal in [97]

illustrated in Fig. 1.1) 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 [30]. In this framework, called the *hierarchical temporal authorization model* (HTAM), no distinction is made between primitive and derived authorizations. This feature required an extension to classical stratification techniques.

The syntax of the policy language is the same as the syntax of TABs [28] with one important difference: the elements of authorization triples can be arbitrary nodes of basic hierarchies. The authorization hierarchy is defined by $(s, o, a, sign, g) \leq (s', o', a', sign, g)$ iff $s \leq s'$, $a \leq a'$ and $o \leq o'$. Conflicts are resolved according to the denial-takes-precedence principle. The formal semantics is formulated by adapting the fixed-point 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 nonstratifiable 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. Stable model existence and uniqueness, as well as its PTIME computability cannot be proved by means of the usual stratification techniques. In [30] the theory of logic programming is extended by identifying a class of nonstratifiable programs—called *almost stratifiable programs*—with the same nice properties as stratifiable programs.

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. These results rely on the denial-takes-precedence principle, that 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, 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), whereas FAF supports multiple such policies. The main goal of FAF is flexibility. So far, no attempt has been made to combine the advantages of both models.

Ordered Logic Programs A significantly different approach, inspired by *ordered logic programs* [102], can be found in [31]. 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*.

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 object 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 [31] propose three different conflict resolution strategies:

1 Use the *well-founded* model of the policy. This (partial) model approximates the intersection of all 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 co-NP-hard problem.

3 Select dynamically a stable model that contains all 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. Powerful engines for computing skeptical and credulous stable model semantics exist [120, 65], 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.

Other approaches KAoS and Ponder support little more than inheritance without exceptions. As we pointed out before, preconditions cannot check whether other authorizations are granted or denied, so rules cannot be chained. Inference consists in matching requests against policies, looking for a more general authorization whose associated preconditions are satisfied. In KAoS, this means computing subsumptions between the given request and the concepts describing authorizations. Overriding is not supported.

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, e.g. by weakening a service request.

This approach is pursued in a series of papers by Minsky et al., including [116, 117]. 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 $\mathsf{send}(s, m, t)$, 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 `send`(s, m, t) onto an action of the form `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 metamessage `^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 run time. 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 [117] 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, the 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, for example, in abductive terms.

Another interesting option is applying a policy description language based on event-condition-action rules, such as $\mathcal{PDL}$ [51, 108], 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.

2. Policy-Based Trust Management

The concept of *trust* has come with many different meanings and it has been used in many different contexts like security, credibility, etc. Work on authentication and authorization allows to perform access control based on the requester's identity or properties. Trust in this sense provides confidence in the source or in the author of a statement. In addition, trust might also refer to the quality of such a statement.

Typically access control systems are identity-based. It means that the identity of the requester is known and authorization is based on a mapping of the requester identity to a local database in order to check if he/she is allowed to perform the requested action. For example, given that Alice asks Bob for access to a resource, she must first authenticate to Bob. This way, Bob can check if Alice should be allowed to access that resource.

Nowadays, however, due to the amount of information and the increase of the World Wide Web, establishment of trust between strangers is needed, i.e., between entities that have never had any common transaction before. Identity-based mechanisms are not sufficient. For example, an e-book store might give a discount to students. In this case, the identity of the requester is not important, but whether he/she is a student or not. These mechanisms are property-based and, contrary to identity-based systems, provide the scalability necessary for distributed environments.

This section offers a historical review of existing research and describe in detail the state of the art of policy-based trust management.

Trust Management

Existing authorization mechanisms are not enough to provide expressiveness and robustness for handling security in a scalable manner. For example, Access Control Lists (ACL) are lists describing the access rights a principal (entity) has on an object (resource). An example is the representation of file system permissions used in the UNIX operating system. However, although ACLs are easy to understand and they have been used extensively, they lack [34]:

- Authentication: ACL requires that entities are known in advance. This assumption might not hold in true distributed environments where an authentication step (e.g. with a login/password mechanism) is needed.
- Delegation: Entities must be able to delegate to other entities (not necessarily to be a Certification Authority) enabling decentralization.

- Expressibility and Extensibility: A generic security mechanism should be extendable with new conditions and restrictions without the need to rewrite applications.

- Local trust policy: As policies and trust relations can be different among entities, each entity must be able to define its own local trust policy.

In order to solve the problems stated above and provide scalability to security frameworks, a new approach called *trust management* [36] was introduced.

In general, the steps a system must perform in order to process a request based on a signed message (e.g. using PGP [154] or X.509 [96]) can be summarized as "Is the key, with which the request was signed, authorized to perform the requested action?". However, some of the steps involved in answering such a question are too specific and can be generalized, integrating policy specifications with the binding of public keys to authorized actions. Therefore, the question can be replaced by "Given a set of credentials, do they prove that the requested action complies with a local policy?". In [36, 34] the "trust management problem" is defined as a collective study of security policies, security credentials and trust relationships. The solution proposed is to express privileges and restrictions using a programming language.

The next subsections describe systems that provided a scalable framework following these guidelines.

PolicyMaker and KeyNote. PolicyMaker [36, 37] addresses the trust management problem based on the following goals:

- Unified mechanism: Policies, credentials, and trust relationships are expressed using the same programming language.

- Flexibility: Both standard certificates (PGP [154] and X.509 [96]) as well as complex trust relationships can be used (with small modifications).

- Locality of control: Each party is able to decide whether it accepts a credential or on whom it relies on as trustworthy entity, avoiding a globally known hierarchy of certification authorities.

- Separation of mechanisms from policies: PolicyMaker uses general mechanisms for credential verification. Therefore, it avoids having mechanisms depending on the credentials or on a specific application.

PolicyMaker consists of a simple language to express trusted actions and relationships and an interpreter in charge of receiving and answering queries. PolicyMaker maps public keys into predicates that represent which actions the

key are trusted to be used for signing. This interpreter processes "assertions" which confer authority on keys.

KeyNote [33, 35] extends the design principles used in PolicyMaker with standardization and ease of integration into applications. Keynote performs signature verification inside the trust management engine while PolicyMaker leaves it up to the calling application. In addition, KeyNote requires credentials to be written in an assertion language designed for KeyNote's compliance checker.

In KeyNote, the calling application sends a list of credentials, policies and requester public keys to the evaluator together with an "action environment". This action environment contains all the information relevant to the request and necessary to make the trust decision. The identification of the attributes, which are required to be included in the action environment, is the most important task in integrating KeyNote into different applications. The result of the evaluation is an application-defined string which is returned to the application.

It is important to note that neither PolicyMaker nor KeyNote enforce policies but give advice to applications that make calls to them. It is up to the calling application whether to follow their advice or not. In addition, although their languages are not rule based, we included them in this section for completion and as a motivation for rule based policy languages.

REFEREE. REFEREE [52] (Rule-controlled Environment For Evaluation of Rules, and Everything Else) is a trust management system that provides policy-evaluation mechanisms for Web clients and servers and a language for specifying trust policies. Its authors define trust as "undertaking a potentially dangerous operation knowing that it is potentially dangerous". The elements necessary to make trust decisions are based on credentials and policies.

REFEREE uses PICS labels [128] as credentials. A PICS label states some properties of a resource in the Internet. In this context, policies specify which credentials must be disclosed in order to grant an action.

In REFEREE credentials are executed and their statements can examine statements made by other credentials and even fetch credentials from the Internet. Policies are needed to control which credentials are executed and which are not *trusted*. The policies determine which statements must be made about a credential before it is safe to run it.

The main difference compared with PolicyMaker [36, 37] is that PolicyMaker assumes that credential-fetching and signature verification are done by the calling application. PolicyMaker receives all the relevant credentials and assumes that the signatures have been already verified before the call to the system.

SD3. SD3 [98] (Secure Dynamically Distributed Datalog) is a trust management system consisting of a high-level policy language, a local policy evaluator and a certificate retrieval system. It provides three main features:

- Certified evaluation: At the same time an answer is computed, a proof that the answer is correct is computed, too.
- High-level language: SD3 abstracts from signature verification and certificate distribution. It makes policies easy to write and understand.
- SD3 is programmable: Policies can be easily written and adopted to different domains.

SD3 language is an extension of datalog. The language is extended with SDSI global names [54]. A rule in SD3 is of the form:

```
T(x,y) :- K$E(x,y) ;
```

In the previous rule, *T(x,y)* holds if a digital credential asserting *E(x,y)* and signed with the private key of *E* was given. Whenever a global name is used, an authentication step is needed. In addition, SD3 can refer to assertions in remote computers. Given the rule

```
T(x,y) :- (K@A)$E(x,y) ;
```

the query evaluator must query a remote SD3 evaluator at an IP address *A*. This gives SD3 the possibility to create "chains of trust".

Trust Negotiation

Often, shared information in traditional distributed environments tells which parties can provide what kind of services and which parties are entitled to make use of those services. Then, trust between parties is a straightforward matter. Even if on some occasions there is a trust issue, as in traditional client-server systems, the question is whether the server should trust the client, and not vice versa. Trust establishment is often handled by uni-directional access control methods, such as having the client log in as a pre-registered user.

In contrast, in new environments like the Web parties may make connections and interact without being previously known to each other. In many cases, before any meaningful interaction starts, a certain level of trust must be established from scratch. Generally, trust is established through exchange of information between the two parties. Since neither party is known to the other, this trust establishment process should be bi-directional: both parties may have sensitive information that they are reluctant to disclose until the other party has proved to be trustworthy at a certain level. As there are more service providers

emerging on the Web every day, and people are performing more sensitive transactions (for example, financial and health services) via the Internet, this need for building mutual trust will become more common.

Trust negotiation is an approach to automated trust establishment. It is an iterative process where trust is established gradually by disclosing credentials and requests for credentials. This differs from traditional identity-based access control and release systems mainly in the following aspects:

- Trust between two strangers is established based on parties' properties, which are proved through disclosure of digital credentials.
- Every party can define access control and release policies (*policies*, for short) to control outsiders' access to their sensitive resources. These resources can include services accessible over the Internet, documents and other data, roles in role-based access control systems, credentials, policies, and capabilities in capability-based systems.
- In the approaches to trust negotiation developed so far, two parties establish trust directly without involving trusted third parties, other than credential issuers. Since both parties have policies, trust negotiation is appropriate for deployment in a peer-to-peer architecture, where a client and server are treated equally. Instead of a one-shot authorization and authentication, trust is established incrementally through a sequence of bilateral credential disclosures.

A trust negotiation is triggered when one party requests to access a resource owned by another party. The goal of a trust negotiation is to find a sequence of credentials $(C_1, \ldots, C_k, R)$, where R is the resource to which access was originally requested, such that when credential C_i is disclosed, its policy has been satisfied by credentials disclosed earlier in the sequence—or to determine that no such credential disclosure sequence exists.

A detailed discussion on general criteria for trust negotiation languages as well as important features (like well-defined semantics, expression of complex conditions, sensitive policies and delegation) can be found in [137].

Regulating Service Access and Information Release. A formal framework to specify information disclosure constraints and the inference process necessary to reason over them and to filter relevant policies given a request is presented in [40]. A new language is presented with the following elements:

- *credential(c,K)* where c is a credential term and K is a public key term.
- *declaration(attribute_name=value_term)*

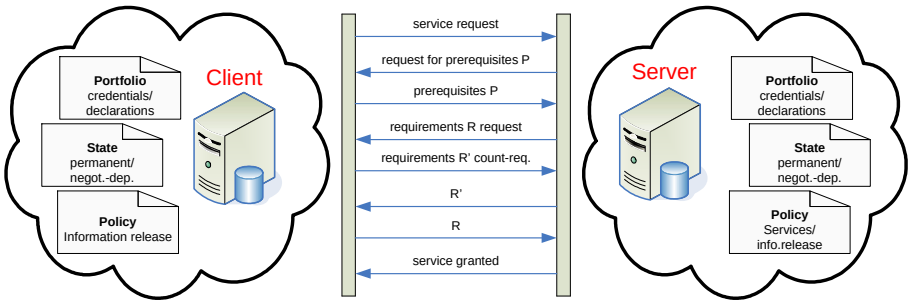

Figure 1.2. Client/Server interplay

- *cert_authorityy(CA,K_{CA})* where CA represents a certification authority and K_{CA} its public key.
- *State predicates* which evaluates the information currently available at the site
- *Abbreviation predicates*
- *Mathematic predicates* like $=, \neq, <$.

Using the elements described above, rules can be specified in order to regulate the negotiation. There are two kind of rules: *service accessibility rules* and *portfolio disclosure rules*. A service is a functionality that a server offers in the form of e.g. an application that a client can execute. A portfolio is the set of properties that a party can disclose during a negotiation in order to obtain access to or offer services. Therefore service accessibility rules specify the requirements that a client must satisfy in order to get access to a service and portfolio disclosure rules specify the conditions a requester must satisfy in order to receive information from the portfolio.

Portfolio requisite rules define required credentials and declarations that other party must satisfy before portfolio information (credentials or declarations) are disclosed.

These basic elements and rules are all needed to perform a negotiation between a server which offers services and a client who wants to consume them. In order to allow the server to select applicable rules a policy filtering mechanism is needed. This mechanism filters the rules related to a specific request from the server's knowledge base. Those selected rules will be then pre-evaluated locally and/or sent to the client. Figure 1.2 shows an example scenario of the interaction process between client and server.

RT: Role-based Trust-Management. The RT framework [106, 104, 105] is a set of languages for representing policies and credentials. It

is specially suited for "decentralized collaborative systems" (systems where they do not have to loose the authority over the resources they control) and for attribute-based access control (ABAC). Those systems must be able to express decentralized attributes, delegation of attribute authority, inference of attributes, attribute field and attribute-based delegation of attribute authority.

RT uses roles in order to represent attributes. An entity has an attribute if it is a member of the corresponding role. The RT framework consists of several components:

RT_0 It is the most basic language of the RT set. It addresses all the requirements described above except "attribute fields".

In RT_0 policy statements take the form of role definitions. Role definitions have a head of the form $K_A.R$ and a body. K_A represents a principal while R is a role term. RT_0 allows constructions for simple membership, simple containment, linking containment, intersection containment, simple delegation and linking delegation.

RT_1 In RT_0 roles do not take any paremeters. RT_1 role definitions have the same form as in RT_0 but they may contain parameterized roles. In RT_1 a role is of the form $r(p_1, \ldots, p_n)$. r is the role name and p_i can be $name = c$, $name = ?X[\in S]$ ($\in S$ is optional) or $name \in S$ where $name$ represents a name of a parameter, c represents a constant, $?X$ is a variable and S is a value set.

RT_2 RT_2 adds to RT_1 logical objects (also o-set) in order to group permissions between objects. A credential in RT_2 is either an o-set-definition or a role-definition. An o-set-definition is formed by an entity followed by an o-set identifier ($K.o(h_1, \ldots, h_n)$ and allows to constrain variables with dynamic value sets (inferred from roles or o-sets).

RT^T Sometimes it is required that two or more different entities are responsible to perform a sensitive task together for its completion. RT^T provides manifold roles and role-product operators. A manifold role defines a set of principals sets. Each of these sets is a set of principals whose collaboration satisfies the manifold role.

RT^D RT^D provides delegation of role activations which express selective use of capacities and delegation of these capacities. A delegation credential presented by a principal D takes the form of $D \overset{D \; as \; A.R}{\longleftarrow} B_0$. With it a principal D activates the role $A.R$ to use in a session B_0. In addition B_0 can further delegate this role activation with $B_0 \overset{D \; as \; A.R}{\longleftarrow} B_1$.

PEERTRUST. PEERTRUST [72, 119] builds upon previous work on policy-based access control and release for the Web and implements automated trust negotiation for such a dynamic environment.

PEERTRUST's language is based on first order Horn rules (definite Horn clauses), i.e., rules of the form "$lit_0 \leftarrow lit_1, \ldots, lit_n$" where each lit_i is a positive literal $P_j(t_1, \ldots, t_n)$, P_j is a predicate symbol, and the t_i are the arguments of this predicate. Each t_i is a term, i.e., a function symbol and its arguments, which are themselves terms. The head of a rule is lit_0, and its body is the set of lit_i. The body of a rule can be empty.

Definite Horn clauses are the basis for logic programs, which have also been used as the basis for the rule layer of the Semantic Web and specified in the RuleML effort ([83, 85]) as well as in the recent OWL Rules Draft [91]. Definite Horn clauses can be easily extended to include negation as failure, restricted versions of classical negation, and additional constraint handling capabilities such as those used in constraint logic programming. Although all of these features can be useful in trust negotiation, here we describe instead other more unusual required language extensions.

References to Other Peers. The ability to reason about statements made by other peers is central to trust negotiation. To express delegation of evaluation to another peer, each literal lit_i is extended with an additional *Authority* argument,

lit_i @ Authority

where *Authority* specifies the peer who is responsible for evaluating lit_i or has the authority to evaluate lit_i. The *Authority* argument can be a nested term containing a sequence of authorities, which are then evaluated starting at the outermost layer.

A specific peer may need a way of referring to the peer who asked a particular query. This is accomplished by including a *Requester* argument in literals, so that now literals are of the form

lit_i @ Issuer $ Requester

The *Requester* argument can be nested, too, in which case it expresses a chain of requesters, with the most recent requester in the outermost layer of the nested term.

Using the *Issuer* and *Requester* arguments, it is possible to delegate evaluation of literals to other parties and also express interactions and the corresponding negotiation process between parties.

Signed Rules. Each peer defines a policy for each of its resources, in the form of a set of definite Horn clause rules. These and any other rules that

the peer defines on its own are its *local* rules. A peer may also have copies of rules defined by other peers, and it may use these rules in its proofs in certain situations.

A signed rule has an additional argument that says who signed the rule. The cryptographic signature itself is not included in the logic program, because signatures are very large and are not needed by this part of the negotiation software. The signature is used to verify that the issuer really did issue the rule. It is assumed that when a peer receives a signed rule from another peer, the signature is verified before the rule is passed to the DLP evaluation engine. Similarly, when one peer sends a signed rule to another peer, the actual signed rule must be sent, and not just the logic programmatic representation of the signed rule. More complex signed rules often represent delegations of authority.

Putting it together. Figure 1.3 depicts an implemented scenario in an e-learning domain [72, 140] (PEERTRUST has also been applied, among other scenarios, to Grid [21] and Semantic Web Services [123]). Alice and E-Learn obtain trust negotiation software signed by a source that they trust (PEERTRUST Inc.) and distributed by PEERTRUST Inc. or another site, either as a Java application or an applet. After Alice requests the Spanish course from E-Learn's web front end, she enters into a trust negotiation with E-Learn's negotiation server. The negotiation servers may also act as servers for the major resources they protect (the Learning Management Servers (LMS)), or may be separate entities, as in our figure. Additional parties can participate in the negotiation, if necessary, symbolized in our figure by the InstitutionA and InstitutionB servers. If access to the course is granted, E-Learn sets up a temporary account for Alice at the course provider's site, and redirects her original request there. The temporary account is invisible to Alice.

Cassandra. Cassandra [23, 24] is a role-based trust management system. It uses a policy language based on datalog with constraints and its expressiveness can be adjusted by changing the constraint domain. Policies are specified using the following predicates which govern access control decisions:

- $permits(e, a)$ specifies who can perform which action
- $canActivate(e, r)$ defines who can activate which role (e is a member of r)
- $hasActivated(e, r)$ defines who is active in which role
- $canDeactivate(e, r)$ specifies who can revoke which role
- $isDeactivated(e, r)$ is used to define automatically triggered role revocation

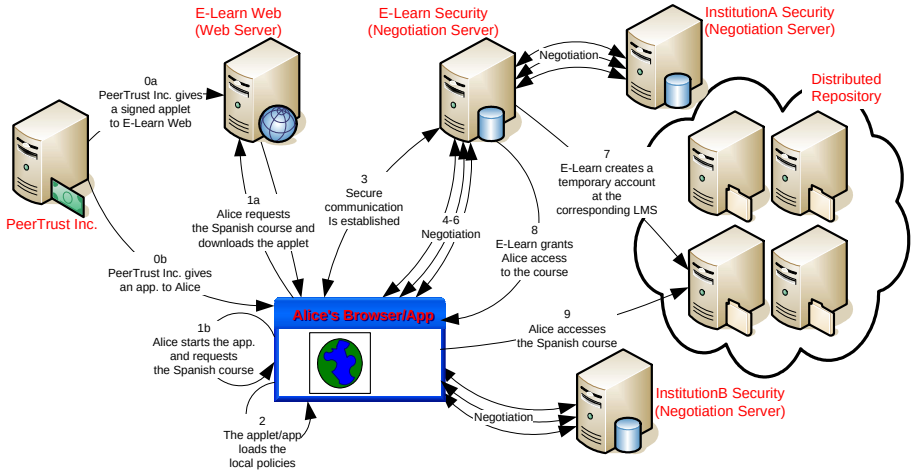

Figure 1.3. PEERTRUST: Automated Trust Negotiation for Peers on the Semantic Web

- $canReqCred(e_1, e_2, p(e))$ specifys the requirements that a request must satisfy in order to issue and disclose credentials

This way, policy managers can define and use new predicates as they need. A Cassandra predicate also contains an *issuer* and a *location* like

loc @ iss.p(e)

where *location* represents the entity where the assertion applies (and therefore it allows queries over the network) and the issuer is the entity that asserts it.

Although Cassandra does not provide special constraints to specify role validity periods, auxiliary roles, role hierarchy, separation of duties, role delegation, automated trust negotiation and credential discovery, it can express these kind of policies. That way the language and its semantics are simpler and makes easier the extension of the language.

A policy rule in Cassandra is of the form:

$$E_{loc}@E_{iss}.p_0(e_0) \leftarrow loc_1@iss_1.p_1(e_1), \ldots, loc_n@iss_n.p_n(e_n), c$$

where p_i are the names of the predicates, e_i is a set of expression tuples and c is a constraint.

A rule with only a constraint c in its body like

$$E_{loc}@E_{iss}.p_0(e_0) \leftarrow c$$

represents a credential signed and issued by E_{iss} asserting $p_0(e_0)$ which is stored at E_{loc}.

PROTUNE. The PRovisional TrUst NEgotiation framework PROTUNE [42] aims at combining distributed trust management policies with provisional-style business rules and access-control related actions. PROTUNE's rule language extends the PAPL [40] and PEERTRUST [72] languages and it provides a powerful declarative metalanguage for driving some critical negotiation decisions, and integrity constraints for monitoring negotiations and credential disclosure. PROTUNEwill be further described in section 5.0.

3. Action Languages

Reasoning about action and change is a kind of temporal reasoning where, instead of reasoning about *time* itself, we reason on *phenomena* that take place in time. *Action theories* are formal theories for reasoning about action and change, that describe a *dynamic world* changing because of the execution of actions. Properties characterizing the dynamic world are usually specified by propositions which are called *fluents*. The word *fluent* stresses the fact that the truth value of these propositions depends on time and may vary depending on the changes which occur in the world.

The problem of reasoning about the effects of actions in a dynamically changing world is considered one of the central problems in knowledge representation theory. Different approaches in the literature took different assumptions on the temporal ontology and they developed different abstraction tools to cope with dynamic worlds. However, most of action theories describe dynamic worlds according to the so-called *state-action model*. In the state-action model the world is described in terms of states and actions that cause the transition from a state to another. More precisely, there are some assumptions that typically hold in action theories referring to the *state-action model*, that we list below:

- the dynamic world to be modeled is always in a determined state;
- change is interpreted as a transition from a world state to another;
- the world persists in its state unless it is modified by an action execution that causes the transition to a new state (*persistency assumption*).

Based on the above conceptual assumptions, the main target of action theories is to use a logical framework to describe the effects of actions on a world where *all* changes are caused by the execution of actions. More precisely, a formal theory for representing and reasoning about actions allows us to specify:

(a) *causal laws*, i.e. axioms that describe domain's actions in terms of their preconditions and effects on the fluents;

(b) action sequences that are executed from the initial state;

(c) *observations* describing the fluent's value in the *initial state*;

(d) *observations* describing the fluent's value after some action execution.

In the following, the term *domain description* is used to refer to a set of propositions that express causal laws, observations of the fluents value in a state and possibly other information for formalizing a specific problem. Given a domain description, the principal reasoning tasks are *temporal projection* (or prediction), *temporal explanation* (or postdiction) and *planning*. Intuitively, the aim of *temporal projection* is to predict action's future effects based on (even partial) knowledge about the actual state (reasoning from causes to effects). On the contrary, the target of *temporal explanation* is to infer something on the past states of the world by using knowledge about the actual situation. The third reasoning task, *planning*, is aimed at finding an action sequence that, when executed starting from a given state of the world, produces a new state where certain desired properties hold.

Usually, by varying the reasoning task, a domain description will contain different elements that provide a basis for inferring the new facts. For instance, when the task is to formalize the temporal projection problem, a domain description will contain information on (a), (b) and (c), then the logical framework will provide the inference mechanisms for reconstructing information on (d). Otherwise, when the task is to deal with the planning problem, the domain description will contain the information on (a), (c), (d) and we will try to infer (b), i.e. which action sequence has to be executed on the state described in (c) for achieving a state with the properties described in (d).

A relevant formalization difficulty is known as the *persistency problem*. It concerns the characterization of the invariants of an action, i.e. those aspects of the dynamic world that are not changed by an action. If a certain fluent f, representing a fact of the world, holds in a certain state and it is not involved by the next execution of an action a, then we would like to have an efficient inference mechanism to conclude that f still holds in the state resulting from the execution of a.

A second formalization difficulty, known as the *ramification problem*, arises in presence of the so-called indirect effects (or ramifications) of actions and concerns the problem of formalizing *all* the changes caused by an action's execution. Indeed, action's execution might cause a change not only on those fluents that represent its direct effects, but also on other fluents which are indirectly involved by the chain of events started by the action's execution.

Various approaches in the literature can be broadly classified in two categories: those choosing classical logics as knowledge representation language [113, 101] and those addressing the problem by using non-classical logics [48, 136, 77] or computational logics [74, 20, 109, 15]. In the following, we will

briefly review the most popular logic-based approaches to reason about action and change.

Expressing policies by means of declarative languages is surely useful because it simplifies the verification of properties; in particular using the action metaphor for modelling policies is one of the most intuitive ways. Different to the standard use of actions, in this case there are further situations that might be taken into account. For example, as suggested in [130], it might be useful to enrich the action description by adding mid-conditions, i.e. conditions that might become true at execution time. The idea is to capture specific conditions that are relevant in modelling security. On the other hand, research in action theory has proposed extensions that deal with reasoning about complex actions that we believe are useful to increase the expressiveness of security policies.

Logical Approaches

Among the various logic-based approaches to reasoning about actions one of the most popular is still the situation calculus, introduced by Mc Carthy and Hayes in the sixties [113]. The situation calculus represents the world and its change by a sequence of *situations*. Each situation represents a state of the world and it is obtained from a previous situation by executing an action. A world is represented as a sequence of actions, called a *situation*, starting from an *initial situation* S_0. A binary function symbol $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. For example, the term $do(putdown(A), do(pickup(A), S_0))$ is a situation denoting the world resulting from the sequence of actions [pickup(A), putdown(A)]. *Fluents*, i.e. relations whose truth values vary from situation to situation, are denoted by predicate symbols taking a situation as their last argument. For example $on(A, B, s)$ means that block A is on block B in situation s. An *action theory* can be defined by giving *preconditions* and *effects* for each action.

As pointed out by McCarthy and Hayes [113] formalizing an action theory requires dealing with *persistency*, by specifying those fluents which remain unaffected by a given action. Since most of the fluents do not change from a state to the next one, we want a parsimonious solution to this problem. The main idea is to minimize change from one state to the next by making use of nonmonotonic logics or of completion constructions. In particular, Reiter [127] proposed a solution which relies on the *completeness assumption* that the action theory describes all action laws affecting the truth value of any fluent f. Under this assumption, it is possible to define a *successor state axiom* for each fluent, giving the value of this fluent in the next state.

Kowalski and Sergot have developed a different calculus to describe change [101], called *event calculus*, in which *events* producing changes are temporally located and they initiate and terminate action effects. Like the situation

calculus, the event calculus is a methodology for encoding actions in first-order predicate logic. However, it was originally developed for reasoning about events and time in a logic-programming setting.

Another approach to reasoning about actions is the one based on the use of modal logics. The suitability of dynamic logics or modal logics to formalize reasoning about actions and change has been pointed out in various proposals [60, 48, 136, 77]. Modal logics adopt essentially the same ontology of situation calculus by taking the state of the world as primary and by representing actions as state transitions. In particular, actions are represented in a very natural way by modalities whose semantics is a standard Kripke semantics given in terms of accessibility relations between worlds, while states are represented as sequences of modalities. In modal logic, a primitive action a can be represented by a modal operator $[a]$, and a sequence of actions $a_1, a_2, \ldots, a_n$ by the modal operator $[a_1; a_2; \ldots; a_n]$ ($[\varepsilon]$ represents the empty sequence of actions, i.e. the initial state). Furthermore we can use a modality $\Box$ to represent an arbitrary sequence of actions. For instance , action effects can be expressed as $\Box[load]loaded$, meaning that fluent $loaded$ holds after execution of action $load$ in any state, or $\Box(loaded \Rightarrow [shoot]\neg alive)$ and $\Box[shoot]\neg loaded$, meaning that after action $shoot$ fluent $loaded$ will be false, and fluent $alive$ will be false if $loaded$ holds before executing the action. Although the representation with modal logic and the one with first-order logic are apparently similar, it is important to point out a significant difference between the two. In fact, if we do not assume any particular property for the modal operators representing actions (modal logic K), the two formulas $\neg[s]\phi$ and $[s]\neg\phi$ have different meanings, whereas in the situation calculus both would be represented by $\neg\phi(s)$. Thus, differently from the situation calculus, we cannot derive $\neg loaded$ from the above rules and $[shoot]alive$, i.e. action rules cannot be used contrapositively.

Computational Logic

Non-classical logics have successfully been used for developing agent theories, for representing and reasoning about action and change as well as for modeling mental attitudes as beliefs, knowledge and goals. This is due to their capability of representing *structured and dynamic knowledge*. However a wide gap between the expressive power of the formal models and the practical implementations has emerged, due to the computational effort required for verifying that properties granted by logical models hold in the systems that implement them.

For this reason there is a growing interest on the use of *computational logic*, which allows one to express formal specifications that can be directly executed, thanks to the fact that logic programs have a procedural interpretation, besides the declarative one [146]. In '93 Gelfond and Lifschitz have defined a sim-

ple declarative language for describing actions, called $\mathcal{A}$ [74]. The target is to define a logical entailment relation between a domain description and simple *queries* of the form "f **after** $a_1, \ldots, a_n$" where f is a fluent and $a_1, \ldots, a_n$ are elementary actions with the meaning of "Is it true that the fluent f holds after the execution of actions $a_1, \ldots, a_n$?". As a difference of the computational logic approach with respect to a logical approach, the domain description contains causal laws for the domain's actions and observations on fluents value in the initial state, and for all of them and for the goal it is easy to give a goal directed proof procedure, that is prolog-like. Alternatively, it is possible to translate the representation directly in Prolog. Actually, the language $\mathcal{A}$ has been formally defined by giving a translation into general logic programming extended with explicit negation. Note that the entailment relation of $\mathcal{A}$ is nonmonotonic, and this aspect is modeled by the *negation as failure* of logic programming.

Various extensions of $\mathcal{A}$ have been proposed in the last years with the intention to deal with nondeterministic actions [100, 17], concurrent actions [19], ramifications [100, 79] or sensing actions [109, 20]. Most of the times, a sound translation of such extensions into logical languages is provided.

Executable agent specification languages

The theory of computational agents plays a central role in Artificial Intelligence, providing powerful conceptual tools for characterizing complex systems situated in dynamic environments [126, 152]. Software agents are usually designed as highly autonomous entities (each agent has an internal state -i.e. a set of attitudes like beliefs and goals- and uses it to take decisions), able to react to variations in their environments (reactivity), able to achieve their goals (proactiveness) by interacting, if necessary, with other agents (social capabilities). Thus, they must be able to perform practical reasoning such as planning, i.e. the process of deciding how to achieve a *goal* using the available means (the actions which can be performed).

One of the core research issues in this field is the definition of formal languages for specifying single agents (including the representation of their knowledge and their reasoning techniques) and for modeling communication and interaction among agents. Many theories about agency are based on *logic formalisms* [126, 56, 152, 53].

Modeling agent's internal behaviour and attitude dynamics is a difficult task. Generally speaking, it is impossible to assume either *knowledge completeness* (agents can have partial and incomplete views on the external world) or *knowledge uniqueness* (different agents can have different views on the external world). In presence of incomplete knowledge, the output of reasoning is not expected to be a fixed sequence of actions (*linear plan*), but a more

general specification involving conditionals, in which the next action to perform depends on the result of a sensing actions (*conditional plan*). An agent could start in the generation of a plan from scratch, as in standard planning in AI. Alternatively, it could be provided with a *library of plans*, which have been predefined by the designer: finding a plan to achieve a goal means extracting from the library a plan that, when executed, will have the goal as a post-condition, and that is sound given the agent's current beliefs. Predefined plans can also be partially specified, by means of procedures, defining complex actions the agent can perform. In this case the planning task is interpreted as finding a terminating execution of the procedure that leads to a state where the desired goal holds. The procedure definition constrains the search space in which to look for the wanted sequence.

In order to cope with these issues, extensions of classical logics and new reasoning techniques have been studied. Most of the approaches build on the top of an action theory expressed in one of the formalisms reviewed in the previous sections. Non-classical logics (like modal logics, deontic logic and non-monotonic logics) have been successfully used for developing agent theories, both to represent and reason about actions, and to formalize mental states and their dynamics. Computational logic is often used to develop *logic-based executable agent specifications* due to its nature it also supports the verification tasks. In fact, in logic programming, logic *is* the programming language and agent programs can be specified as logical rules that can be executed by a SLD-style proof procedure.

Logic-based, executable agent specification languages have been deeply investigated in the last decade [103, 16, 7, 68]. Both situation calculus and modal action logics influenced the design of these languages, in particular, the cognitive robotic project at University of Toronto has lead to the development of a high-level agent programming language, called GOLOG, which is based on *situation calculus* [103]. A *modal action theory* has been used as a basis for specifying and executing agent behaviour in the logic programming language DyLOG [16]. The language IMPACT is an example of use of *deontic logic* for specifying agents. In the following, we will briefly recall the main features of these three languages, which seem to be particularly promising for the implementation of intelligent application systems, including security.

GOLOG is a programming language, for the specification and the execution of complex actions in dynamic domains based on the situation calculus [7]. It is a procedural language mainly designed for programming high-level robot control and intelligent software agents. Recently it has been used for allowing the automatically composition of services on the semantic web in the context of the DAML-S initiative [114, 115]. Primitive actions are specified by giving their preconditions and effects in terms of suitable axioms. Formalization of complex actions refers to an Algol-like paradigm and draws considerably from

dynamic logic. In particular, action operators like sequence, nondeterministic choice and iteration are provided. Second order logic is needed for defining iteration and procedures. Once extracted, a GOLOG plan can be executed by the agent. Thus, it is possible to implement rational agents alternating planning and execution. Specific works in this direction has driven to the development of IndiGolog [75]. The problem of dealing with concurrency instead has been faced in [61, 62], by leading to the definition of CONGOLOG.

Like GOLOG, the language DyLOG is designed for specifying agents behaviour and for modeling dynamic systems. As a main difference, it is fully set inside the logic programming paradigm by defining programs by sets of Horn-like rules and giving a SLD-style proof procedure. DyLOG has been used for implementing a *virtual tutor* [14], that helps students to build personalized study curricula, while in [10, 9, 11] the capability of reasoning about interaction protocols, supported by the language, has been exploited for customizing web service selection and composition w.r.t. to the user's constraints, based on a semantic description of the services. The language is based on a modal theory of actions and mental attitudes where *modalities* are used for representing primitive and complex *actions* as well as the agent beliefs [16]. Complex actions are defined by *inclusion axioms* [8] and by making use of action operators from dynamic logic, like sequence ";", test "?" and non-deterministic choice "$\cup$". DyLOG rules can be used also for giving a local representation of interaction protocols, i.e. for defining conversation policies -building upon FIPA-like speech-acts- that the agent follows when interacting with others.

DyLOG supports planning and temporal projection, by allowing to prove existential properties of the kind "given a procedure p and a set of desiderata, is there a legal sequence of actions conforming to p that, when executed from the initial state, also satisfies the desired conditions?". In case we deal with communicative behavior, this process is meant to find an answer to the query: "given a conversation policy p and a set of desiderata, is there a specific conversation, respecting the policy, that also satisfies the desired conditions". As in GOLOG the procedure definition constrains the search space. A goal-directed proof procedure has been developed that implements such kinds of reasoning and planning, and allows to automatically extract, from DyLOG procedures, *linear* or *conditional* plans for achieving a given goal from an incompletely specified initial state [16, 10].

The IMPACT agent architecture, introduced by Subrahmanian et al. [7], provides a framework to build agents on top of heterogeneous sources of knowledge. To agentize such sources, the authors introduce the notion of agent program. Such agent programs and their semantics resemble logic programs extended with deontic modalities. Indeed, they consist of a set of rules suitable to specify, by means of deontic modalities, agent policies in normative terms, that is what actions an agent is obliged to take in a given state, what actions it

is permitted to take, and how it chooses which actions to perform. Thus, if $\alpha(\vec{t})$ is an action with parameters $\vec{t}$, then $\mathbf{O}\alpha(\vec{t})$, $\mathbf{P}\alpha(\vec{t})$, $\mathbf{F}\alpha(\vec{t})$, $\mathbf{Do}\alpha(\vec{t})$, $\mathbf{W}\alpha(\vec{t})$, are the so-called action status atoms possibly included in IMPACT rules, which are read (respectively) as $\alpha(\vec{t})$ is obligatory, permitted, forbidden, done, and the obligation to do $\alpha(\vec{t})$ is waived. In IMPACT, at every state transition, the agent determines a set of actions to be executed, obeying some notion of deontic consistency. IMPACT has been used to develop real applications ranging from combat information management to aerospace applications.

Reasoning about interaction

Communication and dialogue have intensively been studied in the context of formal theories of agency [93]. In particular, great attention has been devoted to the definition of *standard agent communication languages* (ACL), such as FIPA and KQML. The crucial issue was to achieve interoperability in open agent systems, characterized by the interaction of heterogeneous agents; to this aim it is fundamental to have a universally shared semantics.

Agent communication languages are complex structures because a communicative act must specify many kinds of information. The definition of a formal semantics for individual communicative acts has been one of the major topics of research in this field. Most of the proposals are based on the philosophical theory of speech acts developed by Austin and Searle in the sixties: communications are not merely considered as the transmission of information but as *actions* that, instead of modifying the external world, affect the mental states of the involved agents. Thus, ACL semantics of individual speech acts is given in terms of preconditions and effects on the mental attitudes (as it is commonly done with action semantics). Standard techniques for reasoning about change can be exploited for proving conversation properties, for planning communication with other agents and for answer selection as in [44, 67]. Therefore, ACL-like speech act they can naturally be represented in agent programming languages based on action theories, e.g. the languages DyLOG.

Opposed to the mentalistic approach, followed by ACL-based agent languages as DyLOG, some authors have recently proposed a *social approach* to agent communication [139], in which communicative actions affect the "social state" of the system rather than the internal states of the agents. The social state records the social facts, like the *permissions* and the *commitments* of the agents, which are created and modified along the interaction. The birth of the social approach is due to the difficulty of verifying, in a mentalistic framework, that an agent acts according to a commonly agreed semantics, because its mental state cannot be accessed [149], a problem known as *semantics verification*. The social approach overcomes the semantics verification problem because it

exploits a set of established commitments *between* the agents, that are stored as part of the MAS social state.

Recently the attention has been moved towards the formalization of those aspects of communication that are related to the conversational context in which communicative acts occur [111] with the introduction of *conversation policies* and *interaction protocols*. The need for the verification if a given property holds for a given protocol has recently emerged as a fundamental requirement. To such purpose formal languages and analysis tools are currently considered aspects of primary importance, especially for security protocols, where formal proofs of properties are indeed fundamental for the protocol itself [80]. The research on protocol verification has greatly benefit from some important contributions achieved in the distributed and concurrent systems research area. In particular, the results obtained in model checking [55, 27] have been proved extremely useful for the verification of protocols. A notable example is the SPIN system [89, 90] where interacting entities can be defined as finite state automata through PROMELA (PROtocol LAnguage Goal), and protocol properties can be expressed through formulas in temporal linear logic. In the area of agent languages based on logic, some examples of definition of protocols for guiding the agent communicative behavior can be found [66, 13, 12]. The logical formalization supports the definition of elegant techniques for conformance verification of agent policies w.r.t. public protocol specifications.

The basic idea is that protocols built upon a predefined set of speech acts. The social approach provides a high-level specification of the protocol, and does not require the rigid specification of all the allowed action sequences by means of finite state diagrams, which is instead typical of mental approaches. In a social framework it is possible to formally prove the correctness of public interaction protocols with respect to the specifications outcoming from the analysis phases; such proof can be obtained, for instance, by means of model checking techniques [125, 149, 78, 26]. However, when one passes from the public protocol specification to its *implementation* in some language (e.g. Java, DyLOG), a program is obtained which, by definition, relies on the information contained in the internal "state" of the agent for deciding which action to execute. In this perspective, the use of a declarative language is helpful because it allows the proof of properties of the *specific implementation* in a straightforward way. In particular, the use of a language that explicitly represents and uses the agent internal state is useful for proving to which extent certain properties depend on the agent mental state or on the semantics of the speech acts. For instance, in [16, 10, 9, 11] the hypothetical reasoning about the effects of conversations on the agent mental state is used to find conversation plans which are proved to respect the implemented protocols, achieving at the same time some desired goal.

4. Business Rules

Business rules are "statements about how a business is done, i.e. about guidelines and restrictions with respect to states and processes in an organization" [25]; they "formulate a law or custom that guides the behavior or actions of the actors connected to the organization" [145]. Business rules can be formalized and explicitly managed, but they are often implicitly captured in corporate documents, spreadsheets, workflow descriptions and information systems, scattered all over the organization.

While many business rules are (implicitly) introduced from external sources like the culture or the law, in many other cases, business rules are negotiated between the members of the organizations or their representatives. The goal of writing down those rules is to gain reliability and predictable operations of the organization.

Galbraith summarizes the main reasons in favor of using formalized rules as follows [71]: (a) *Coordination*: In complex situations, the execution of tasks may need synchronization of the work done by several persons; (b) *Precision*: The execution of as far as possible formalized tasks is precise over time, i.e. everybody knows what to do in every considered event; (c) *Efficiency*: A machine(like) consistency of the task execution may lead to more efficient production, like in the automobile industry; and (d) *Fairness*: Especially government organizations have to secure an equal treatment of every client; hence they strive to formalize their behavior in order to protect clients and also employees.

Today, a number of business rules are explicitly written down, commonly in organizational handbooks which contain systematically collected and specified business rules; they describe the static and dynamic aspects of an organization: the positions within the organization, verbal descriptions of rights and duties of the employees, and the business processes, illustrating the tasks do be accomplished, their dependencies, time restrictions, and responsibilities.

Another, usually smaller and more homogeneous subset of the business rules of an organization is captured by information systems, which often are also used to *enforce* those rules. Clearly, not all business rules are candidates of being explicitly written down and formalized. Schmidt [135] has established a series of criteria which may help to decide which types of rules and tasks are eligible of formalization. Rules that fit those criteria can be operationalized by transforming them into executable rule expressions, e.g. into a declarative logic based rule language.

Typology of Formalized Business Rules

In this section we are particulary concerned about the explicit formulation of business rules. We follow the top-level classification illustrated by [142] and

[147], which bases on [47] and distinguish three families of business rules: Reaction rules, derivation rules and integrity constraints. In the following, we will characterize these types of rules and their components.

Reaction Rules. Reaction rules are concerned with the invocation of actions in response to events. They state the conditions under which actions must be taken. They define the behavior of a system (or an agent) in response to perceived environment events and to communication events [147]. Reaction rules, often called ECA (Event-Condition-Action) rules, are conceptually of the following type:

```
ON event
IF condition is fulfilled
THEN perform action
```

This concept assumes an *event controller*, which monitors certain types of events, and upon occurrence of such an event, the condition of the rule is evaluated. If the condition is true, the action associated to the rule is executed.

Production Rules. Production rules are similar to ECA rules; they may even be considered a special case of the general concept of reaction rules [147]. In rule based systems, productions rules are of the form `IF C THEN A`, where `C` is a condition and `A` is *any* kind of action, including external procedures/methods.

The inputs to production rule systems are a set of such production rules, i.e. condition-action pairs of the form if *condition* then *action*. The other two components of a production rule system are: (i) *Working memory*: The memory holds the description of the current state of the world in a reasoning process. Most production systems allow to create networks of objects, defined by object templates which have one head and one or more slots (i.e. attribute fields). (ii) *Recognize-act cycle*: The conditions (i.e. left hand side of the rules) are continuously matched against the known facts in the working memory. If one rule applies, it is fired, that is, its right hand side is executed. If more than one rules apply, the conflicting rules are added to a goal agenda, ordered and then executed sequentially. This cycle continues until all rules are satisfied.

The fact that production systems are responsible for determining the set of applicable rules at a given time relieves the programmer (rule modeler) from considering and codifying all the paths by which a rule may become applicable or inapplicable.

The most efficient algorithm for implementing such production systems is the Rete algorithm [69]. The Rete algorithm is the only known algorithm for production systems whose performance is demonstrably independent of the number of rules in the system. An algorithm similar to Rete is TREAT [118],

which differs in several aspects of the organization of the internal working memory of the algorithms.

Derivation Rules. Another class of rules that is widely used for the specification of formal business rules are derivation rules. Derivation rules allow to *derive* knowledge from other knowledge by an inference or a mathematical calculation [147].

Each rule expresses the knowledge that if one set of statements happens to be true, then some other set of statements must also be true (or become true). Using a set of such rules, it is possible to specify the behavior of systems by means of logical specifications. This leads us to the term *Logic Programming* [107], which is a well-known programming paradigm based on a subset of First Order Logic, named Horn clause Logic.

The most prominent example of a language exploiting the advantageous properties of Horn clauses is *Prolog*. Prolog departs from pure logics by supporting numerous extra-logical features, e.g. numeric operations and the CUT. The CUT is a construct that can be used to steer the SLD resolution process.

A variant of Prolog, *Datalog* [49], is used to implement deductive database systems. These systems are called *deductive*, because they are able to deduce new facts from the data already stored in the database.

Datalog is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language [58]. A deductive database uses two main types of specifications: facts and rules. Facts can be compared to relations in RDBM systems, while rules can be compared to SQL views. One of the fundamental differences to SQL views, however, is that Datalog based views (i.e. rules), may involve recursion and hence may yield virtual relations that cannot be defined in terms of standard relational views [58].

Integrity Constraints. An integrity constraint is an assertion that must be satisfied in all evolving state and state transition histories of an enterprise viewed as a discrete dynamic system [142].

In the literature, the following types of integrity constraints are mentioned:

- *State constraints*: These constraints must hold at any point in time. An example of a state constraint is “a customer of the car rental company EU-Rent must be at least 25 years old” [142].

- *Structural assertions*: An important type of state constraints are structural assertions [88]. A structural assertion is a statement that something of importance to the business either exists as a concept of interest or exists in relationship to another thing of interest. It details a specific, static

aspect of the business, expressing things known or how known things fit together.

- *Process constraints*: These refer to the dynamic integrity of a system; they restrict the admissible transitions from one state of the system to another. An process constraint may, for example, declare that the admissible state changes of a RentalOrder object are defined by the following transition path: $reserved \rightarrow allocated \rightarrow effective \rightarrow dropped-off$ [142].

Integrity constraints can be found in many different systems and use very different notations; Constraints can be expressed as IF-THEN statements in programming languages, as explicit assertion statements supported by programming languages such as C++, Eiffel or in the recent Java 2 version 1.5, as CHECK and CONSTRAINT clauses in SQL table definitions and as CREATE ASSERTION statements in SQL database schema definition[142], c.f. also Section "Rules in Active DBMS" below. Finally, structural assertions can be modeled as UML or entity/relationship diagrams and can be augmented by state constraints represented as OCL (Object Constraint Language)

Integrity constraints can also be seen as a special case of ECA rules, because they perform a certain *action* (e.g. repair the database) in the *event* of a violated integrity constraint.

Implementation of Business Rules

Rules in Active DBMS. *Active* DBMS on the other hand, use rules – mainly based on the ECA paradigm – to describe activities to be carried out by the system. Active DBMS monitor events and then react appropriately; hence, active databases present a *reactive* behavior (compared to the passive behavior of typical DBMS): they execute not only user transactions, but also the rules specified.

Many commercial relational systems like Oracle, DB2 Sybase offer this functionality, in the form of triggers (standardised in SQL-3); other examples for active relational DBMS are Ariel [87], Postgres [141] and Starbust [150]. There also exists object oriented active databases such as HiPac [59], Sentinel [50] and EXACT [63].

Besides the reactive behavior, modern database systems are able to capture and enforce another type of rules, i.e. integrity constraints. Constraints are declarations of conditions about the database that must remain true. These include attributed-based, tuple-based and referential integrity constraints. The database system checks for the violation of the constraints on actions that may cause a violation and aborts the action accordingly.

Rule-Based Programming Environments. In the following we briefly review popular rule engines.

Mandarax. Mandarax [64] is an open source java library for business rules. This includes the representation, persistence, exchange, management and processing (querying) of rule bases. The main objective of Mandarax is to provide a pure object oriented platform for rule-based systems.

In Mandarax, rules are presented as clauses that consist of a body (the prerequisite or antecedent of the rule) and a head (i.e. the consequence of the rule). The prerequisites and the conclusion are *facts*, which themselves consist of *terms* and *predicates* associating those terms. Under the object oriented notation supported by Mandarax, terms represent objects while predicates on the other hand represent relationships between terms. Terms can be constants, variables or complex terms; complex terms are terms that can be computed from other terms (functions).

The Mandarax engine uses an object oriented version of backward chaining mechanism similar to Prolog; this is in contrast to popular rule engines like ILOG or JESS, which use the forward chaining Rete algorithm [69]. The Mandarax project offers several rule engines which slightly differ in some implementation aspects (e.g. support of Prolog-like Cut, negation as failure).

The Mandarax project is also developing a reactive variant of the Mandarax rule engine. The engine – called Mandarax ECA – is an extension that can be used to program reactive agents; events have registered event listeners (handlers), these listeners query the knowledge base for the next action that must be performed.

ILOG. ILOG [94] is a rule engine and programming library that allows developers to combine rule-based and object-oriented programming to add business rules to new and existing applications. The ILOG rule engine is exposed to Java[1] and C++ code via an application programmer interface (API). Rules can be dynamically added, modified, or removed from the engine on the fly, i.e. without shutting down or recompiling the application.

ILOG uses an optimized variant of the Rete algorithm, which makes it capable of handling large numbers of rules within an application and achieving a high performance in handling rules. Further, ILOG offers a wide range of enhancements, such as automatic rule optimizations which occur transparently to the developer, auto hashing and indexing.

ILOG rules employs the ILOG Rule Language, which has a Java-like syntax and a variety of language extensions. Developers have at their disposal full

[1]The engine for Java is branded *JRules*.

support of operators in expressions and tests, Java-like syntax for interfaces, arrays, and variable scope management.

The ILOG Rules rule engine can directly parse and output rules in an XML representation, allowing the management of rules by standard XML tools. Further, the ILOG tool suite offers a point-and-click editor to manipulate the rule base.

Jess. Jess [70] is a Java based rule engine and scripting environment inspired by the CLIPS [76, 129] expert system shell with its OPS5 [46] production rule language. Just like Mandarax, Drooles and ILOG, Jess is augmented by an object oriented language (i.e. Java) to increase its applicability for commercial projects (which often have a large legacy code base to support).

Jess can directly make use and manipulate Java objects. Moreover, it is a reference implementation for the JSR-94 standardization proposal, which aims at providing a uniform Java application programming interface to rule engines.

Like CLIPS, Jess is based on the Rete algorithm [69], the forward chaining mechanism for production rule systems. Like all production rule-based systems, the functionality of Jess is comprised of the rule base, the working memory and the recognize-act cycle (cf. Section "Production Rules").

Rule Markup Language (RuleML). RuleML [39, 95] is a standardization initiative that was started in 2000 with the goal to establish an open, vendor neutral XML based rule language standard, permitting both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks.

RuleML foresees a classification of the rule it supports. RuleML encompasses a hierarchy of rules, from reaction rules, via integrity constraints and derivation rules to facts (i.e. premiseless derivation rules). For these top-level families, XML DTDs are provided, reflecting the structures of the rule families.

In the first two years of RuleML, the emphasis has been on the expression of derivation rules XML. Another goal of RuleML is to integrate the rule markup language with ontology languages like DAML+OIL and subsequently OWL. The current outcome of these efforts is a draft for SWRL (Semantic Web Rule Language) [92], which is based on a combination of the OWL DL and OWL Lite sub-languages of OWL with the Unary/Binary Datalog sublanguages of RuleML.

Another goal has been to provide an object oriented extension to rule modeling, as already showcased by several rule engines (c.f. Section "Implementation of Rules in Information Systems"). To date (Summer 2004) there exists a system of XML DTDs for *slotted* (i.e. frame-based) RuleML sublanguages including the Object-Oriented RuleML (OO RuleML) [38]. Recent efforts also

went into defining MOF-RuleML [148], the abstract syntax of RuleML as an MOF Model and aligning RuleML with UML's Object Constraint Language (OCL).

A critical review of RuleML is given in [147]. One of the weaknesses identified by that paper is the lack of support of ECA rules. This limitation is currently being addressed by a working devoted to Reactive RuleML [1].

Dealing with Rule Conflicts and Inconsistency

Conflicts Among Rules - Causes. In many logical systems, like Horn logic, there can be no conflicts between rules: once the premises of a rule are satisfied, the rule is executed and its conclusion is drawn. This is due to the fact that negation in the rule heads is not allowed.

Once we allow negation to appear in the rule head, the situation becomes more complicated because it is possible that two rules may lead to contradictory conclusions. Conflicting rules are not necessarily indications of an error in the knowledge base, but may arise naturally in different ways.

Conflicting rules are useful as a modeling feature. For example, rules with exceptions, found in many policies, can be expressed naturally using a set of conflicting rules: a rule describing the general case, and rules expressing exceptions. For example, the general rule may say that all professors are tenured, while an exception rule may say that visiting professors are not tenured.

Another type of application scenarios is reasoning with incomplete information. In these scenarios, the available knowledge is insufficient to mace certain decisions, but we have to make conclusions based on "rules of thumb". A typical scenario is emergency medical diagnosis, where initial diagnosis and treatment needs to be made before the results of medical tests become available. Note that new information may lead to a revision of the initial decisions. These scenarios are closely linked to the area of nonmonotonic reasoning [112, 3].

Conflicting rules also naturally arise in knowledge integration, when knowledge from different sources (and possibly authors) is combined. This scenario is expected to be particularly wide-spread on the Semantic Web, where a key idea is to import knowledge from various sources and adapt it for own purposes.

Dealing with Conflicting Rules. The question is how to deal with situations where rules with conflicting heads can potentially be applied. In first-order logic and related approaches, contradictory conclusions may be drawn but have trivialization effects: every conclusion can be drawn from a contradictory set of premises.

This behaviour is deemed to be unacceptable for practical purposes. It considers contradictions as error situations, but we explained previously that this is not necessarily the case.

Reasoning systems falling in this category are wide-spread in logic programming, knowledge representation and the Semantic Web [2, 4, 82, 127].

One way of resolving conflicts is to use priorities among rules. For example, the rule stating that visiting professors are not tenured is stronger than the rue stating that professors are tenured. With this information incorporated in the knowledge base, the conclusion that a particular visiting professor is not tenured can be drawn even sceptically. The aforementioned works make use of such priorities.

Obviously, we need ways of incorporating priorities in the reasoning process even in complex cases. An extensive body of work is available in this directions [2, 4, 82, 121, 134]. We should also mention work on developing rules systems tailored to the Semantic Web that are able of dealing with inconsistent and incomplete information, among them [84, 22].

The Origin of Priorities. Priorities may arise from internal or external sources. Internal priorities are computed from a set of rules based on the idea of specificity: a more specific rule is viewed as an exception to a more general rule and should therefore be deemed to be stronger. For a system that computes priorities based on specificity of rules see [32].

While useful, specificity is only one prioritization principle. To capture other principles, most logical systems rely on priorities that are made available externally. That is, priorities are considered to be a part of the knowledge base, as are rules and facts. External priority information may be based on a number of principles:

- One rule may be preferred to another rule because it is an exception to another rule. Such information is often stated explicitly in policies and business rules [5].
- One rule may be preferred to another because it is more recent. This principle is often used in law and regulations.

Apart from these principles which apply to pairs of individual rules, priority information may be based on comparing groups of rules. For example, in business administration the rules originating from higher management have higher authority than those originating from middle management. Or in knowledge integration, one source of rules may be known to be more reliable than the other. This preference of groups is propagated to individual rules.

5. Unifying Frameworks

Cleary all specifications we have described in the previous sections interact tightly with each other. Trust management policy languages need to express security policies and actions, business rules and action languages describe how

"things are done", security specifications play an important role for business rules and decisions. While it is hard or maybe even impossible to really integrate all aspects described in the previous chapters in one framework, unifying several of these aspects in one framework is necessary for comprehensive applications of rule-based policy specifications, and makes explicit the interaction between the various features treated separately in many previous approaches. In this section, we will therefore describe two different approaches towards a unifying framework: XACML and PROTUNE.

XACML

The eXtensible Access Control Markup Language (XACML) [122] is an OASIS standard that describes both a policy language and an access control decision request/response language (both in XML).

The policy language allows to specify access control conditions that must be fulfilled by a requester. There are three kind of top-level elements:

<**Rule**> It is a boolean expression which is not intended to be evaluated in isolation but can be reused by several policies.

<**Policy**> It is a set of rules and obligations that apply to a request. It contains a set of rules and an algorithm describing how to combine the results of their evaluation.

<**PolicySet**> It contains a set of policy and policy set elements together with an algorithm describing how to combine the results of their evaluation.

The request/response language allows to send queries in order to check whether a specific request should be allowed. There are four different valid values for the answer in the response: Permit, Deny, Indeterminate (a decision could not be made) or Not Applicable (the request can't be answered by this service)

They provide the basis for the separation of the so called Policy Enforcement Point (PEP) which is the entity in charge of protecting a resource and the Policy Decision Point (PDP) which is responsible for checking whether a request is conformant with a given policy. In order to include the execution of actions within the standard, the authors define the <*Obligation*> element. An obligation is "an action that must be performed in conjunction with the enforcement of an authorization decision". In the current version of XACML 2.0 there are no standard definitions for these actions.

XACML is a standard, so it includes many features among which we higlight the following:

- The language allows the use of attributes in order to perform authorization decisions without relying exclusively on the identity of requester.

- Different arithmetic, set and boolean operators, and built-in functions are provided as well as a method to extend the language with non-standard functions.

- The language includes a <*Target*> element in each rule, policy or policy set in order to allow indexing and increase performance.

- Different combination algorithms are provided for rule and policy composition: `deny-overrides`, `ordered-deny-overrides`, `permit-overrides`, `ordered-permit-overrides`, `first-applicable` and `only-one-applicable`.

- An *XACML context* is define in order to provide a canonical form for representing requests and responses. As it is encoded in XML, it is possible to extract information from the context using XPath 2.0.

However, the current specification of XACML (v2.0 at the time being) is not suitable as a policy language for protocols like trust negotiation (see section 2.0). It still lacks some required expressivity like delegation of authority (see [138] for a discussion on requirements for trust negotiation) which have to be taken into consideration for next versions of the specification.

Protune

The PRovisional TrUst NEgotiation framework PROTUNE [42] aims at combining distributed trust management policies with provisional-style business rules and access-control related actions. PROTUNE's rule language extends two previous languages: PAPL [40], that until 2002 was one of the most complete policy languages for trust negotiation [137], and PEERTRUST [72], that supports distributed credentials and a more flexible policy protection mechanism. In addition, the framework features a powerful declarative metalanguage for driving some critical negotiation decisions, and integrity constraints for monitoring negotiations and credential disclosure.

PROTUNE provides a framework with:

- A trust management language supporting general provisional-style actions (possibly user-defined).

- An extendible declarative metalanguage for driving decisions about request formulation, information disclosure, and distributed credential collection.

- A parameterized negotiation procedure, that gives a semantics to the metalanguage and provably satisfies some desirable properties for all possible metapolicies.

- Integrity constraints for negotiation monitoring and disclosure control.
- General, ontology-based techniques for importing and exporting metapolicies and for smoothly integrating language extensions.

The PROTUNE rule language is based on normal logic program rules "$A \leftarrow L_1, \ldots, L_n$" where A is a standard logical atom (called the *head* of the rule) and $L_1, \ldots, L_n$ (the *body* of the rule) are literals, that is, L_i equals either B_i or $\neg B_i$, for some logical atom B_i.

A *policy* is a set of rules, such that negation is applied neither to *provisional predicates* (defined below), nor to any predicate occurring in a rule head. This restriction ensures that policies are *monotonic* in the sense of [137], that is, as more credentials are released and more actions executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [18].

The vocabulary of predicates occurring in the rules is partitioned into the following categories: *Decision Predicates* (currently supporting "allow()" which is queried by the negotiation for access control decisions and "sign()" which is used to issue statements signed by the principal owning the policy, *Abbreviation/Abstraction Predicates* (as described in [40]), *Constraint Predicates* (which comprise the usual equality and disequality predicates) and *State Predicates* (which perform decisions according to the state). State Predicates are further subdivided in *State Query Predicates* (which read the state without modifying it) and *Provisional Predicates* (which may be made true by means of associated actions that may modify the current state like e.g. $credential(C, K), declaration(), logged(X, logfile\_name)$).

Furthermore, metapolicies consist of rules similar to object-level rules. They allow to inspect terms, check groundness, call an object-level goal G against the current state (using a predicate $holds(G)$), etc. In addition, a set of reserved attributes associated to predicates, literals and rules (e.g., whether a policy is public or sensitive) is used to drive the negotiator's decisions. For example, if p is a predicate, then p.`sensitivity` : `private` means that the extension of the predicate is private and should not be disclosed. An assertion p.`type` : `provisional` declares p to be a provisional predicate; then p can be attached to the corresponding action α by asserting p.`action` :α. If the action is to be executed locally, then we assert p.`actor` : `self`, otherwise assert p.`actor` : `peer`.

6. Summary and Open Research Issues

The term *policy* is used with different meanings in the existing literature but generally refers to the "specification of the behaviour of a system". The

most important aspect of policy specification is the language issue, i.e. which language to use to express those policies. In this chapter we provided an exhaustive overview on rule based policy specification encompassing security and trust management policies, action languages and business rules.

Security Policies pose constraints on the behaviour of a system. They are typically used to control permissions of users/groups while accessing resources and services. The languages for specifying security policies are more and more focussing on logic-based, and especially rule-based languages. After exploring a number of different constructs (such as negation as failure, temporal operators, deontic modalities, etc.) and different semantics (top down, tabled, abductive, and dynamic, like event-condition-action rules) it is clear that such languages are extremely flexible and capable of capturing the most diverse policies arising in real application scenarios.

The term trust management refers to reputation based metrics and models, or to policy based trust. *Trust Management policy languages* are used to collect user properties in open environments, where the set of potential users spans the entire web. In this section we offered an extensive review of existing research on policy-based trust management. Among other properties, all current languages have in common that they do not consider authorization as a one-shot process anymore but instead rely on trust negotiations in order to establish trust between strangers.

Action Languages are used in reactive policy specification to execute actions like event logging, notifications, etc. Authorizations that involve actions and side effects are sometimes called provisional. Action languages also allow to reason about phenomena that take place in time and to describe a dynamic world changing because of the execution of actions. Proposals based on (classical and non-classical) logics and computational logic are the most successful ones. Besides the representation used, an action theory is formed by a set of action laws that describe actions in term of preconditions and effects on the world. Typical kinds of reasoning performed on an action theory are temporal projection, temporal explanation, and planning.

Business Rules are "statements about how a business is done" and are used to formalize and automatize business decisions as well as for efficiency reasons. We provided an overview of formal languages and approaches for expressing such rules, to gain readability and predictable operation of an organization. In particular, we provided a typology of formalized business rules and described the main approaches.

All these kinds of specification interact tightly with each other and new approaches are appearing aiming at their unification in a single language/framework.

However, although the research community has achieved great advances in the area, there still exist several open issues and challenges for policy specifi-

cation. Some of the main research issues concern *integration*, *ease of use* and *implementation*.

A major integration issue concerns the harmonization of the different semantics mentioned above (top down, tabled, etc.), when the policy comprises both declarative and ECA rules. The problem is even more subtle in trust negotiation, where action execution must be scheduled appropriately during multiple negotiation steps.

Integration is also related to implementation. The powerful rule-based policy languages being developed might have a fast and widespread impact if we could (sometimes) translate high-level rules into policies supported by common mechanisms (such as firewalls and the access control mechanisms of DBMSs and web servers). Then high-level policy specifications would let security managers organize their policies in a homogeneous and coherent way, without giving up the efficiency and robustness of lower-level security mechanisms. It would be possible to have a centralized, global view of the system's policy without necessarily introducing bottlenecks such as centralized security monitors.

One of the few open representation problems that has not yet been extensively explored concerns *delegation*: what can a peer do with a piece of information it receives? These policies should express in a simple and compact way the requirements which the information owner poses on subsequent disclosures, as well as allow the receiver to add its own constraints (when possible) [99, 119, 153]. Expressing this kind of dynamic behavior appropriately in a declarative way, accessible to a vast class of users, is a nontrivial challenge. The same is true of the development of a (cryptographic?) infrastructure capable of enforcing this kind of policies, that otherwise would be left to voluntary compliance. Another issue is integration with trust and reputation models [140, 41] and with other security approaches and new applications (e.g., Grids [131, 21]).

On the other hand, in the context of business rules, formal representation is still an open problem. Work will continue on developing formal languages combining sufficient expressive power, efficient reasoning support, and naturalness of expression.

Last but not least, it will be important to give common users the ability of understanding and personalizing their systems' policies. This is essential to bring the existing security mechanisms to their full potential, and increasing user awareness about security problems. Adopting a rule language is not enough for this purpose, because common users typically have no knowledge about logics (and nonmonotonic logics in particular). It is necessary to provide user-friendly front-ends that illustrate the policy in a language familiar to the user, such as a graphical language, or maybe natural language. Actually, there are already lines of research pursuing the formulation of policy rules in a con-

trolled fragment of natural language. Moreover, very recent work is tackling explanation mechanisms that support advanced queries to policies [43]. Such an explanation tool is meant to guide the user in acquiring the permissions necessary to get the desired services. This kind of support is crucial in e-business applications: a *cooperative way of enforcing policies* may be the key to success in such application scenarios.

Acknowledgements

The authors would like to thank the contribution of Cristina Baroglio, Alberto Martelli, and Viviana Patti to the Section 3.

References

[1] A. Adi, Z. Sommer, A. Biger, S. Ross-Talbot, and G. Wagner. Reactive ruleml, `http://groups.yahoo.com/group/reactive-ruleml/`, 2004.

[2] J. Alferes and L. Pereira. *Reasoning with logic programming*, volume 1111 of *LNAI*. Springer-Verlag, 1996.

[3] G. Antoniou. *Nonmonotonic Reasoning*. The MIT Press, 1997.

[4] G. Antoniou, D. Billington, G. Governatori, and M. Maher. Representation results for defeasible logic. *ACM Transactions on Computational Logic*, 2:255–287, 2001.

[5] G. Antoniou, D. Billington, and M. Maher. On the analysis of regulations using defeasible rules. In *Proc. of HICSS'99*, 1999.

[6] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.

[7] K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V. Subrahmanian. IMPACT: a platform for collaborating agents. *IEEE Intelligent Systems*, 14(2):64–72, 1999.

[8] M. Baldoni. Normal Multimodal Logics with Interaction Axioms. In D. Basin, M. D'Agostino, D. M. Gabbay, S. Matthews, and L. Viganò, editors, *Labelled Deduction*, volume 17 of *Applied Logic Series*, pages 33–53. Applied Logic Series, Kluwer Academic Publisher, 2000.

[9] M. Baldoni, C. Baroglio, L. Giordano, A. Martelli, and V. Patti. Reasoning about communicating agents in the semantic web. In F. Bry, N. Henze, and J. Maluszynski, editors, *Proc. of the 1st International Workshop on Principle and Practice of Semantic Web Reasoning, PPSWR 2003*, volume 2901 of *LNCS*, pages 84–98, Mumbai, India, December 2003. Springer.

[10] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about self and others: communicating agents in a modal action logic. In *Proc. of ICTCS'2003*, volume 2841 of *LNCS*, pages 228–241. Springer, 2003.

[11] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. In M. Bravetti and G. Zavattaro, editors, *Proc. of 1st Int. Workshop on Web Services and Formal Methods, WS-FM 2004*, volume 105 of *Electronic Notes in Theoretical Computer Science*, pages 21–36. Elsevier Science Direct, 2004.

[12] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Verification of protocol conformance and agent interoperability. In F. Toni and P. Torroni, editors, *Proc. of Sixth International Workshop on Computational Logic in Multi-Agent Systems, CLIMA VI*, London, UK, June 2005.

[13] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying protocol conformance for logic-based communicating agents. In J. Leite and P. Torroni, editors, *Post Proc. of Fifth International Workshop on Computational Logic in Multi-Agent Systems, CLIMA V*, LNAI. Springer, 2005.

[14] M. Baldoni, C. Baroglio, and V. Patti. Web-based adaptive tutoring: an approach based on logic agents and reasoning about actions. *Artificial Intelligence Review*, 22(1):3–39, 2004.

[15] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. D. et al., editor, *Proc. of NMELP'96*, volume 1216 of *LNAI*, pages 132–150. Springer-Verlag, 1997.

[16] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming Rational Agents in a Modal Action Logic. *Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation*, 41(2-4):207–257, 2004.

[17] C. Baral. Reasoning about actions: non-deterministic effects, constraints, and qualification. In *Proc of IJCAI'95*, pages 2017–2023, 1995.

[18] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, Cambridge, 2003.

[19] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. Journal of Logic Programming, 31(1-3):85–117, May 1997.

[20] C. Baral and T. C. Son. Formalizing Sensing Actions - A transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001.

[21] J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett. Negotiating trust on the grid. In *2nd WWW Workshop on Semantics in P2P and Grid Computing*, New York, USA, may 2004.

[22] N. Bassiliades, G. Antoniou, and I. Vlahavas. DR-DEVICE: A defeasible logic system for the semantic web. In *Proc. 2nd International Workshop on Principles and Practice of Semantic Web Reasoning*, LNCS. Springer Verlag, 2004.

[23] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, Yorktown Heights, June 2004.

[24] M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop*, Pacific Grove, CA, June 2004.

[25] J. Bell, D. Brooks, E. Goldbloom, R. Sarro, and J. Wood. Knowledge representation, reasoning and declarative problem solving. Technical report, US West Information Technologies Group, Bellevue Golden, 1990.

[26] J. Bentahar, B. Moulin, J. J. C. Meyer, and B. Chaib-Draa. A computational model for conversation policies for agent communication. In J. Leite and P. Torroni, editors, *Pre-Proc. of CLIMA V*, pages 66–81, Lisbon, Portugal, September 2004.

[27] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

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

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

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

[31] E. Bertino, E. Ferrari, F. Buccafurri, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proc. of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 175–189. IEEE Computer Society, 1999.

[32] D. Billington, K. de Coester, and D. Nute. A modular translation from defeasible nts to defeasible logics. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 151–177, 1990.

[33] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System Version 2. In *Internet Draft RFC 2704*, Sept. 1999.

[34] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. *Lecture Notes in Computer Science*, 1603:185–210, 1999.

[35] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols Workshop*, Cambridge, UK, 1998.

[36] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[37] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *Financial Cryptography*, British West Indies, Feb. 1998.

[38] H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. Object-Oriented RuleML, version 0.85 of 15 march 2004,`http://www.ruleml.org/indoo`, 2004.

[39] H. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for semantic web rules. In *International Semantic Web Working Symposium (SWWS)*, 2001.

[40] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *CCS '00: Proceedings of the 7th ACM conference on computer and communications security*, pages 134–143. ACM Press, 2000.

[41] P. A. Bonatti, C. Duma, D. Olmedilla, and N. Shahmehri. An integration of reputation-based and policy-based trust management. In *Semantic Web Policy Workshop in conjunction with 4th International Semantic Web Conference*, Galway, Ireland, nov 2005.

[42] P. A. Bonatti and D. Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 14–23, Stockholm, Sweden, jun 2005. IEEE Computer Society.

[43] P. A. Bonatti, D. Olmedilla, and J. Peer. Advanced policy queries. Project Deliverable D4, Working Group I2, EU NoE REWERSE, Sept. 2005.

[44] P. Bretier and D. Sadek. A rational agent as the kernel of a cooperative spoken dialogue system: implementing a logical theory of interaction. In J. Müller, M. Wooldridge, and N. Jennings, editors, *Intelligent Agents III, proc. of ECAI-96 Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, volume 1193 of *LNAI*. Springer-Verlag, 1997.

[45] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[46] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Series In Artificial Intelligence. Addison-Wesley, 1985.

[47] J. Bubenko, D. Brash, and J. Stirna. Ekd - enterprise knowledge development user guide, 1998.

[48] M. Castilho, O. Gasquet, and A. Herzig. Modal tableaux for reasoning about actions and plans. In S. Steel, editor, *Proc. ECP'97*, LNAI, pages 119–130, 1997.

[49] S. Ceri, G. Gottlob, and L. Tanca. Logic programming and databases. In *Surveys in Computer Science*, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[50] S. Chakravarthy, E. Anwar, L. Maugis, and D.Mishra. Design of Sentinel: An object-oriented DBMS with event-based rules. *Information and Software Technology*, 9:559–568, 1994.

[51] J. Chomicki, J. Lobo, and S. Naqvi. A logic programming approach to conflict resolution in policy management. In *Proc. of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 121–132. Morgan Kaufmann, 2000.

[52] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *World Wide Web Journal*, 2:127–139, 1997.

[53] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. Expressing collaboration and competition among abductive logic agents. In K. Satoh and F. Sadri, editors, *CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA-00)*, 2000.

[54] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in spki/sdsi. *Journal of Computer Security*, 9(4):285–322, 2001.

[55] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[56] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[57] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. Technical report, Imperial College, October 2000.

[58] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 1995.

[59] U. Dayal, B. Blaustein, A. Buchmann, and S. Chakravarthy. The HiPAC project: Combining active databases and timing constraints. In *ACM SIGMOD*, pages 51–70, 1988.

[60] G. De Giacomo and M. Lenzerini. PDL-based framework for reasoning about actions. In *Proc. of AI*IA '95*, volume 992 of *LNAI*, pages 103–114, 1995.

[61] G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of IJCAI'97*, pages 1221–1226, Nagoya, August 1997.

[62] G. De Giacomo and H. J. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. In *Proceedings of the AAAI 1998 Fall Symposium on Cognitive Robotics*, Orlando, Florida, USA, October 1998.

[63] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

[64] J. Dietrich. The mandarax manual, `http://mandarax.sourceforge.net/docs/mandarax.pdf`, 2003.

[65] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for non-monotonic reasoning. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97*, volume 1265 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1997.

[66] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Logic-based agent communication protocols. In F. Dignum, editor, *Advances in agent communication languages*, volume 2922 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 91–107. Springer-Verlag, 2004.

[67] FIPA. FIPA 2000. Technical report, FIPA (Foundation for Intelligent Physical Agents), November 2000.

[68] M. Fisher. A survey of concurrent metatem - the language and its applications. In D. Gabbay and H. Ohlbach, editors, *Proc. of the First International Conference on Temporal Logic, ICTL '94*, volume 827 of *LNAI*, pages 480–505. Springer-Verlag, July 1994.

[69] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37, 1982.

[70] E. Friedman-Hill. *Jess in Action*. Manning Publications Co., 2003.

[71] J. Galbraith. *Organization Design*. Addison-Wesley, 1997.

[72] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st European Semantic Web Symposium (ESWS 2004)*, volume 3053 of *Lecture Notes*

in Computer Science, pages 342–356, Heraklion, Crete, Greece, may 2004. Springer.

[73] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[74] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[75] G. D. Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardina. On the semantic of deliberation in Indigolog: from theory to implementation. In *Proc. of KR 2002*, pages 603–614. Academic Press, 2002.

[76] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming, 3rd Edition*. PWS Publishing Co., Boston, MA, USA, 1998.

[77] L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. ECAI-98*, pages 537–541, 1998.

[78] L. Giordano, A. Martelli, and C. Schwind. Verifying communicating agents by model checking in a temporal action logic. In *JELIA'04*, volume 3229 of *LNAI*, pages 57–69, Lisbon, Portugal, 2004. Springer-Verlag.

[79] E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing actions: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[80] D. Gollman. Analysing security protocols. In A. E. Abdallah, P. Ryan, , and S. Schneider, editors, *Proc. of FASec 2002*, volume 2629 of *LNCS*, pages 71–80. Springer-Verlag, 2003.

[81] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[82] B. Grosof. Prioritized conflict handling for logic programs. In *International Symposium on Logic Programming (ILPS-97*, 1997.

[83] B. Grosof. Representing e-business rules for the semantic web: Situated courteous logic programs in RuleML. In *Proceedings of the Workshop on Information Technologies and Systems (WITS)*, New Orleans, LA, USA, Dec. 2001.

[84] B. Grosof, M. Gandhe, and T. Finin. Sweetjess: Translating damlruleml to jess. In *International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, held in conjunction with the First International Semantic Web Conference (ISWC-2002)*, 2002.

[85] B. Grosof and T. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings of the 12th World Wide Web Conference*, Budapest, Hungary, May 2003.

[86] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

[87] E. Hanson. Rule condition testing and action execution in ariel, 1992.

[88] D. Hay and K. Healy. Defining business rules – what are they really?, 2000.

[89] G. Holzmann. *Description and Validation of Computer Protocols*. Prentice Hall, 1992.

[90] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[91] I. Horrocks and P. Patel-Schneider. A proposal for an owl rules language. http://www.cs.man.ac.uk/ horrocks/DAML/Rules/, Oct. 2003.

[92] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. A semantic web rule language combining OWL and RuleML, version 0.5 of 19 november 2003, `http://www.daml.org/2003/11/swrl/`, 2003.

[93] H. Huget, editor. *Communication in Multiagent Systems*, volume 2650 of *LNAI*. Springer, 2003.

[94] ILOG web site, http://www.ilog.com, 2004.

[95] R. Initiative. The Rule Markup Initiative Web Site, `http://ruleml.org/`, 2004.

[96] International Telecommunication Union. *Rec. X.509 - Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, Aug. 1997.

[97] S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmanian. Flexible supporting for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[98] T. Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[99] G. Karjoth, M. Schunter, and M. Waidner. The platform for enterprise privacy practices - privacyenabled management of customer data, 2002.

[100] G. N. Kartha and V. Lifschitz. Actions with Indirect Effects (Preliminary Report). In *Proc. of the KR'94*, 1994.

[101] R. Kowalski and M. Sergot. A Logic-based Calculus of Events. *New Generation of Computing*, 4:67–95, 1986.

[102] N. Leone and P. Rullo. Ordered logic programming with sets. *J. Log. Comput.*, 3(6):621–642, 1993.

[103] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *J. of Logic Programming*, 31:59–83, 1997.

[104] N. Li and J. Mitchell. RT: A Role-based Trust-management Framework. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, Washington, D.C., Apr. 2003.

[105] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 2002.

[106] N. Li, W. Winsborough, and J. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, Feb. 2003.

[107] J. Lloyd. *Logic Programming*. Springer Verlag, 1984.

[108] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 291–298. AAAI Press, 1999.

[109] J. Lobo, G. Mendez, and S. R. Taylor. Adding Knowledge to the Action Description Language $\mathcal{A}$. In *Proc. of AAAI'97/IAAI'97*, pages 454–459, Menlo Park, 1997.

[110] J. J. Lu, G. Moerkotte, J. Schü, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD Conference*, pages 340–351, 1995.

[111] A. Mamdani and J. Pitt. Communication protocols in multi-agent systems: A development method and reference architecture. In *Issues in Agent Communication*, volume 1916 of *LNCS*, pages 160–177. Springer, 2000.

[112] V. Marek and M. Truszczynski. *Nonmonotonic Reasoning*. Springer Verlag, 1993.

[113] J. McCarthy and P. Hayes. Some, Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1963.

[114] S. McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, Trento,Italy, 2002.

[115] S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, 2001.

[116] N. Minsky and D. Rozenshtein. A software development environment for law-governed systems. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*. P.B. Henderson (ed.), 1988.

[117] N. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS 1998)*, pages 322–331. IEEE Computer Society, 1998.

[118] D. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of the National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, pages 42–47, 1987.

[119] W. Nejdl, D. Olmedilla, and M. Winslett. Peertrust: Automated trust negotiation for peers on the semantic web. In *VLDB Workshop on Secure Data Management (SDM)*, volume 3178 of *Lecture Notes in Computer Science*, pages 118–132, Toronto, Canada, aug 2004. Springer.

[120] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[121] D. Nute. *Handbook of Logic for Artificial Intelligence and Logic Programming, Vol. III)*, chapter Defeasible logic, pages 353–395. Oxford University Press, 1994.

[122] extensible access control markup language (XACML) version 2.0. oasis standard, feb 2005.

[123] D. Olmedilla, R. Lara, A. Polleres, and H. Lausen. Trust negotiation for semantic web services. In *1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, volume 3387 of *Lecture Notes in Computer Science*, pages 81–95, San Diego, CA, USA, jul 2004. Springer.

[124] L. Palopoli and C. Zaniolo. Polynomial-time computable stable models. *Ann. Math. Artif. Intell.*, 17(3-4):261–290, 1996.

[125] L. R. Pokorny and C. R. Ramakrishnan. Modeling and verification of distributed autonomous agents using logic programming. In *Pre-Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, pages 172–187, 2004.

[126] A. Rao and M. Georgeff. Modeling rational agents whithin a bdi-architecture. In *Proceedings of KR'91*, pages 473–484, 1991.

[127] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[128] P. Resnick and J. Miller. PICS: Internet access controls without censorship. *Communications of the ACM*, 39(10):87–93, Oct. 1996.

[129] G. Riley. Clips - a tool for building expert systems, web site, `http://www.ghg.net/clips/clips.html`, 2004.

[130] T. Ryutov and C. Neuman. The Specification and Enforcement of Advanced Security Policies. In *Proc. of the Conference on Policies for Distributed Systems and Networks, POLICY 2002*, Monterey, California, June 5–7 2002.

[131] T. Ryutov, L. Zhou, C. Neuman, T. Leithead, and K. E. Seamons. Adaptive trust negotiation and access control. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 139–146, New York, NY, USA, 2005. ACM Press.

[132] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, pages 205–217, 1990.

[133] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[134] T. Schaub and K. Wang. A semantic framework for preference handling in answer set programming. *Logic Programming Theory and Practice*, 3:569–607, 2003.

[135] J. Schmidt. Planvolle steuerung gesellschaftlichen handelns. *Verlag für Sozialwissenschaften*, 1983.

[136] C. B. Schwind. A logic based framework for action theories. In J. Ginzburg et al., editor, *Language, Logic and Computation*, pages 275–291. CSLI, 1997.

[137] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, Monterey, CA, June 2002. IEEE Computer Society.

[138] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation, 2002.

[139] M. P. Singh. A social semantics for agent communication languages. In *Proc. of IJCAI-98 Workshop on Agent Communication Languages*, Berlin, 2000. Springer.

[140] S. Staab, B. K. Bhargava, L. Lilien, A. Rosenthal, M. Winslett, M. Sloman, T. S. Dillon, E. Chang, F. K. Hussain, W. Nejdl, D. Olmedilla, and V. Kashyap. The pudding of trust. *IEEE Intelligent Systems*, 19(5):74–88, 2004.

[141] M. Stonebraker, E. Hanson, and C. Hong. The design of the postgres rule system. In *3rd International IEEE Conference on Data Science*, 1987.

[142] K. Taveter and G. Wagner. Agent-oriented enterprise modeling based on business rules. In *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, pages 527–540. Springer-Verlag, 2001.

[143] D. Toman, J. Chomicki, and D. S. Rogers. Datalog with integer periodicity constraints. In *SLP*, pages 189–203, 1994.

[144] A. Uszok, J. M. Bradshaw, and R. Jeffers. Kaos: A policy and domain services framework for grid computing and semantic web services. In *Trust Management, Second International Conference, iTrust 2004, Oxford, UK, March 29 - April 1, 2004, Proceedings*, volume 2995 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2004.

[145] F. van Assche. Information systems developement: a rule-based approach. *Knowledge-based Systems*, 4:227–234, 1988.

[146] M. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4), 1976.

[147] G. Wagner. How to design a general rule markup language? In *XML Technologien für das Semantic Web - XSW 2002, Proceedings zum Workshop*, pages 19–37. GI, 2002.

[148] G. Wagner, S. Tabet, and H. Boley. MOF-RuleML: The abstract syntax of RuleML as a MOF model. In *Integrate 2003*, 2003.

[149] C. Walton. Model checking agent dialogues. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT'04)*, volume 3476 of *LNAI*. Springer-Verlag, 2005.

[150] J. Widom and S. Ceri. *Active Database Systems: triggers and rules for advanced database processing*. Morgan Kaufmann Publishers Inc., 1996.

[151] 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.

[152] M. Wooldridge and N. R. Jennings. Agent Theories, Architectures, and Languages: A survey. In *Proc. of the ECAI-94 Workshop on Agent Theories*, volume 890 of *LNAI*, pages 1–39. Springer-Verlag, 1995.

[153] C. Zhang, M. Winslett, and P. A. Bonatti. Peeraccess: A logic for distributed authorization. In *12th ACM Conference on Computer and Communication Security (CCS 2005)*, Alexandria, VA, USA, nov 2005. ACM Press.

[154] P. Zimmerman. *PGP User's Guide*. MIT Press, 1994.