I2-D1

Rule-based Policy Specification: State of the Art and Future Work

Project number:	IST-2004-506779
Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Document type:	D (deliverable)
Nature of document	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Naples/I2-D1/D/PU/b1
Responsible editor(s):	P. A. Bonatti
Reviewer(s):	G. Wagner
Contributing participants:	Hannover, Heraklion, Linköping, Naples, St-Gallen, Turin, Zurich, Enigmatec
Contributing workpackages:	I2
Contractual date of delivery:	31 August 2004

Abstract
This report provides 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. It identifies usage scenarios for rule based policies in a semantic web context and it outlines the possible directions of future research.

Keyword List
semantic web, reasoning, security, trust, reputation, action languages, business rules, Attempto Controlled English, application scenarios

Rule-based Policy Specification: State of the Art and Future Work

Piero A. Bonatti[1], Nahid Shahmehri[2], Claudiu Duma[2], Daniel Olmedilla[3], Wolfgang Nejdl[3], Matteo Baldoni[4], Cristina Baroglio[4], Alberto Martelli[4], Viviana Patti[4], Paolo Coraggio[1], Grigoris Antoniou[5], Joachim Peer[6], Norbert E. Fuchs[7]

[1] Dipartimento di Scienze Fisiche, Universit'a di Napoli, Complesso Universitario di Monte SantAngelo, Via Cinthia, I-80126, Napoli, Italy
Email: {bonatti, Paolo.Coraggio}@na.infn.it
[2] Laboratory for Intelligent Information Systems, Dept. of Computer and Information Science, Linköping University, 581 83 Linköping, Sweden
Email: {nahsh, cladu}@ida.liu.se
[3] L3S Research Center and Hannover University, Deutscher Pavillon Expo Plaza 1, 30539 Hannover, Germany
Email: {olmedilla, nejdl}@l3s.de
[4] Dipartimento di Informatica, Università degli Studi di Torino, Italy
Email: {baldoni, baroglio, mrt, patti}@di.unito.it
[5] Information Systems Laboratory, Institute of Computer Science, FORTH, Greece
Email: antoniou@ics.forth.gr
[6] Institute for Media and Communications Management, University of St. Gallen, Switzerland
Email: joachim.peer@unisg.ch
[7] Department of Informatics, University of Zurich, Switzerland
Email: fuchs@ifi.unizh.ch

4 September 2004

Abstract
This report provides 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. It identifies usage scenarios for rule based policies in a semantic web context and it outlines the possible directions of future research.

Keyword List
semantic web, reasoning, security, trust, reputation, action languages, business rules, Attempto Controlled English, application scenarios

Contents

Part I

State of the Art

Introduction

For a long time, logic programming and rule-based formalisms have been considered appealing policy specification languages, as witnessed by a large body of literature.

The most common type of policies are *security policies*, which are used to pose constraints on a systems's behavior (such as *file F cannot be accessed by user U*), but they can also be seen as specification of more complex behavior, including decisions (*what should be asked to user U before granting access to service S?*) and explanations (such as suggesting how to get the permissions to obtain the desired service). Such features are essential in an open scenario such as the semantic web, where the clients or users of a service are often occasional and do not know much about how to interact with the service.

More recently, the notion of policy has been generalized to include other specifications of behavior and decisions, including business rules in all their forms (integrity, derivation, and reaction rules). In the emerging area of service oriented computing, the word "policy" is sometimes used to refer to the orchestration of elementary and compound services. In this broad sense, policies specify the interplay (dialogs, negotiations, etc.) between different entities and actors, for the purpose of delivering services while enforcing some desired application contraints and client requirements.

Here are some of the potential advantages of representing policies with explicit rule-based representations of their semantics. Writing rules is usually faster and cheaper than writing imperative or OO code. The level of abstraction of rules facilitates their expression in user-friendly languages such as controlled natural language. Rules are more concise and easier to understand, share and maintain, especially in a global open environment such as the web, where self-documenting specifications are one of the current approaches to enabling interoperability.

In a similar perspective, a single declarative (semantic) policy specification can be used in several ways, for example not only to enforce a security policy, but also to enable negotiations and explanations. The connection to the semantic web vision is clear: a knowledge-based definition of the policy can be reused in a variety of ways that need not be figured out in advance, thereby achieving a level of flexibility such as those required by modern interoperability scenarios.

In this broad view, the following topics are definitely or potentially relevant to the design and implementation of rule-based policy languages:

- *Logic-based policy languages.* There is a conspicuous number of approaches to logic-based policy specification, taking into account different aspects of policy specification.

- *Trust management.* The notion of trust is getting more and more attention in the area of secure open systems. Part of it is not (currently) formalized with rules.

- *Action languages.* Policies may have to execute actions (e.g., saving certain requests into log files, activating registration procedures, etc.) Thus logic-based languages for specifying actions are relevant to policy language design.
- *Business rules.* As pointed out before, the notion of policy encompasses business rules.
- *Controlled natural language.* In order to let untrained users learn easily how to craft their own policies, one promising approach consists in adopting a natural language front end. The need for precision (an obvious requirement for a policy specification) can be tackled by adopting a *controlled* fragment of natural language.

In this document, we survey the state of the art in the above areas. This deliverable is meant to support the forthcoming policy language design phase.

Chapter 1

Security Policies

1.1 Logic-based policy specification languages

The size and the complexity of real world security policies makes it impossible to handle "flat" policy representations such as plain authorization lists. Policy complexity is increased by the interplay of multiple, heterogeneous requirements, that must be harmonized and merged into a coherent policy.

- In complex organizations, different branches or departments may have direct control over their own data, and establish their own security policy. At the organization level, these different policies must be merged.
- Frequently, different organizations share part of their data, and must agree on a disclosure policy. For example, national statistical institutes typically distribute data owned by data collectors. So the distributor and the owner must agree on a security policy, merging the requirements of both.
- National laws are an *external* source of security constraints. In particular, privacy laws (currently enforced in different forms by many countries) require sensitive personal data to be protected from any use not explicitly authorized by the owner.
- Even within a single, homogeneous system, different groups of users and different classes of objects may be treated according to different principles.
- A related issue is that there is a tradeoff between protection and system usability, especially when the system is open to the Internet. One of the most famous examples is the security model of Java. The initial strict sandbox model soon turned out to be too rigid for the intended applications of Java, and a finer grained model, based on a more expressive policy language, has been adopted for Java 2 [31].

A further degree of complexity is introduced by the temporal dimension. The authorizations granted by a policy may change along time.

Logic-based policy specification languages usually aim at providing language constructs that enhance clarity, modularity, and other desirable properties. The semantics of policy languages determine the *extension* of each policy (i.e., the set of authorizations granted by the policy).

Logic languages are particularly attractive as policy specification languages. One obvious advantage lies in their clean and unambiguous semantics, suitable for implementation validation, as well as formal policy verification. Second, logic languages can be expressive enough to formulate all the policies introduced in the literature. The declarative nature of logic languages yields a good compromise between expressiveness and simplicity. Their high level of abstraction, very close to the natural language formulation of the policies, makes them simpler to use than imperative programming languages, especially for people with little or no technical training (such as typical security managers). However, such people are not experts in formal logics, either, so generality is sometimes traded for simplicity. For this reason, some languages do not adopt a first-order syntax, even if the policy language is then interpreted by embedding it into a first-order logic (e.g., [4, 17]).

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

The target logic is typically *nonmonotonic*, that is, the set of consequences of a theory does not increase monotonically with the set of axioms in the theory. Policy specification (beyond the realm of security) has been proposed long ago as an application of nonmonotonic logics [45]. The reason is that sometimes decisions must be made in the absence of information. So, when new information is added to the theory, some decision may have to be *retracted* (because they have lost their justification), thereby inducing a nonmonotonic behavior. In the area of security, such default decisions arise naturally in real world policies. For example, *open* policies, prescribe that by default authorizations are granted, while *closed* policies prescribes that they should be denied unless stated otherwise. It will be shown later that a particular form of nonmonotonic reasoning (inheritance with overriding) is useful for incremental and compact policy specification.

It is well known that logic programs with negation-as-failure can be regarded as a fragment of major nonmonotonic logics such as *default logic* [52] and *autoepistemic logic* [48]. Logic programs with negation-as-failure may have multiple canonical models called *stable models* [28]. 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 [7]. For instance, the models of a policy may correspond to all the possible ways of assigning permissions that preserve a *Chinese Wall* policy [18]. The system may then choose a secure permission assignment dynamically, trying to satisfy user requests. In general, the set of alternative models may grow exponentially, and the problem of finding one of them is NP-complete, but there are exceptions with polynomial complexity [54, 50].

Some other authors believe that security managers would not trust the system's automatic choice, and adopt restrictions such as *stratifiability* [2] to guarantee that the canonical model be unique.[1] Such restrictions, yield PTIME semantics at the same time.

[1]Note that the Chinese Wall policy can be expressed also with stratified programs [35].

A nonmonotonic logic has been proposed for the first time as a policy specification language by Woo and Lam [62]. They show how default logic can be used to express a number of different policies. For example, if authorizations are expressed with a predicate $\texttt{auth}(\texttt{subj}, \texttt{action}, \texttt{obj})$ and $\top$ is a tautology, then the following default rules express open and closed policies, respectively.

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

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

Woo and Lam provide also an axiomatization of the Bell-LaPadula security model [3], refined with the *need-to-know* principle.

In order to address complexity issues (inference in default logic is at the second level of the polinomial 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.

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

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

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

1.1.1 Dynamic policies

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

Policy modifications can be specified with an imperative language, but then security managers need some training in programming. A streamlined such language has been investigated by Harrison, Ruzzo and Ullman in [34]. Since an arbitrary number of users and objects can be created with the language, the policy extension has no fixed finite bound, and can encode the tape of an arbitrary Turing machine. Consequently, in this framework, the behavior of the policy extension is undecidable.

Temporal authorization bases

In [4] the sets of users and objects are fixed, and the temporal validity of authorizations is specified through *periodic expressions* and suitable *temporal operators*. The resulting language is not completely general, but it has three major advantages: it is expressive enough for typical real-world policies, it does not require expertise in modal logic (actually, temporal expressions and operators are "hidden" behind natural-language-like expressions), and its inferences are decidable. In other words, the generality of temporal operators is traded for usability and decidability.

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

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

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

A graphical representation of the semantics of the different temporal operators is given in Figure 1.1. Note that WHENEVER corresponds to classical implication. For instance, a rule can state that in the summer of year 2002, summer-staff can read a document WHENEVER regular-staff can read it. ASLONGAS embodies a classical implication and a temporal operator. ASLONGAS derives `A` until $\mathcal{A}$ becomes false for the first time. For instance, with ASLONGAS one can formulate a rule stating that regular-staff can read a document every working day in year 2002, until the first working day in which summer-staff is allowed to read that document. Finally, UPON works like a trigger. For instance, a rule can state that Ann can read pay-checks each working day starting from the first working day in year 2002 in which Tom can write pay-checks.

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

$(\texttt{[begin,end]}, \texttt{P}, \texttt{A}^+)$:
$valid(t, \texttt{A}^+) \leftarrow \texttt{t}_b \leq t \leq \texttt{t}_e$, CNSTR$(\texttt{P}, t)$, **not**$(denied(t, \texttt{s}(\texttt{A}^+), \texttt{o}(\texttt{A}^+), \texttt{m}(\texttt{A}^+))$

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

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

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

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

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

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

Auxiliary clauses:

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

$\{$CNSTR$(\texttt{P}, t) \leftarrow t \equiv_{periodicity(\texttt{P})} y\}$
$\forall y$ such that $t \equiv_{periodicity(\texttt{P})} y \Rightarrow t \in \Pi(\texttt{P})$

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

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

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

Table 1.1: Semantics of periodic authorizations and rules [4]

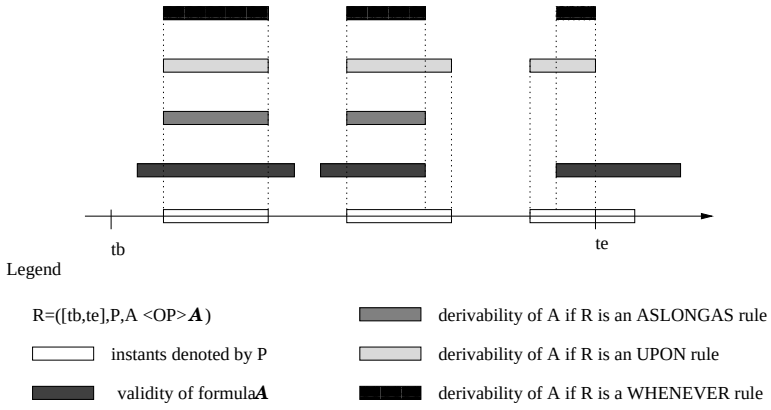


Figure 1.1: Semantics of the different temporal operators [4]

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

Temporal expressions are translated into equations and disequations called *periodicity* and *order constraints*. This mathematical formalism is more appropriate for automated deduction as well as for proving formal properties on the specifications. TABs are translated into sets of Datalog$^{\mathbf{not},\equiv Z,<Z}$ clauses as summarized in Table 1.1.[2]

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

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

In order to implement TAB-based access control efficiently, the canonical model of the corresponding logic program is materialized. 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 [4] 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 [33] and Stdel [44] approaches.

The TABs framework embodies a fixed strategy for conflict resolution (denials-take-

[2]In the table, $\mathtt{A}_i$ is used as a shorthand for an authorization 5-tuple. Expressions $\mathtt{A}_i^-$ and $\mathtt{A}_i^+$ force the sign to be negative and positive respectively. The formulas $\mathcal{A}$ occurring after temporal operators are assumed to be in disjunctive normal form.

precedence). The problem of specifying different strategies (see Section 1.1.2) is not addressed in [4].

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

Active rules

An intermediate approach between imperative and declarative dynamic policy specifications can be found in [6]. The specification language is based on active rules called *role triggers*, whose head specifies actions that modify the policy extension. One difference between this language and previous approaches is that dynamic changes concern *roles*, rather than individual authorizations. Mathematically, a role can be regarded as a relation between users and permissions [57], so by activating and deactivating roles, active rules simultaneously handle entire groups of authorizations. Moreover, roles are relatively static, as they typically correpond to the organization's structure—rules can only enable and disable them, role creation and deletion are not supported. This feature makes dynamic policies decidable, even if the sets of users and objects are unbounded.

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 body is satisfied fire in parallel and schedule the event in their head. The bodies can be made true by previously scheduled events, by events requested at runtime by the security administrator, and by *periodic events*, that is, prioritized events labelled with a periodic expression of the same form as those adopted in [4] (and illustrated previously).

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

Role triggers are more difficult to use than the temporal constraints of TABs (because of side effects), but the former can be naturally implemented through the standard triggers supported by several DBMS (a prototype implementation based on Oracle is described in [6]). Periodic events are materialized like TABs, by considering only the stabilization phase and one period.

The materialization, called *agenda*, is then used to generate events that activate triggers, that are in one-to-one correspondence with the active rules of the policy language. In this way, the policy language gives an abstract and cleaner view of the underlying procedural mechanism supported by the DBMS. As a particular example, the syntactic restrictions introduced in the paper make the effects of the actions independent from the order in which triggers are fired.

1.1.2 Hierarchies, inheritance and exceptions

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

Typically, groups (resp. classes) need not be disjoint and can be nested. The only constraint is that no group (class) should be a member of itself. Roles can be structured in a similar way, although role hierarchies reflect specialization rather than set inclusion [57]. This common structure can be abstracted by the notion of *hierarchy* [35]. Hierarchies are triples $(\mathsf{X}, \mathsf{Y}, \leq)$ where:

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

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

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

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

Exceptions make inheritance a *defeasible* inference, in the sense that inherited authorizations can be retracted (or *overridden*) as exceptions are introduced. As a consequence, the underlying logic becomes 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 uncomparable authorizations. Conflicting inheritance may arise even if the basic hierarchies are trees. For

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

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

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

Jajodia et al. [35] worked on a proposal for a logic-based language that attempted to balance flexibility and expressiveness on the one side, and easy management and performance on the other. The language allows the representation of different policies and protection requirements, while at the same time providing understandable specifications, clear semantics, and bearable data complexity. Their proposal for a Flexible Authorization Framework (FAF) corresponds to a polynomial (quadratic) time data complexity fragment of default logic.

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

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

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

- The authorization table is viewed as a database.
- Policies are expressed by a restricted class of stratified and function-free normal logic programs called *authorization specifications*.
- The semantics of authorization specifications is the stable model semantics [28]. The structure of authorization specifications guarantees stratification, and hence stable model uniqueness and PTIME computability.

Stratum	Predicate	Rules defining predicate
0	`hie`-predicates `rel`-predicates `done`	base relations. base relations. base relation.
1	`cando`	body may contain `done`, `hie-` and `rel`-literals.
2	`dercando`	body may contain `cando`, `dercando`, `done`, `hie-`, and `rel-` literals. Occurrences of `dercando` literals must be positive.
3	`do`	in the case when head is of the form `do`$(\_, \_, +a)$ body may contain `cando`, `dercando`, `done`, `hie-` and `rel-` literals.
4	`do`	in the case 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 1.2: Rule composition and stratification of the proposal in [35]

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 (`rel`-predicates), and additional application-specific predicates, called `rel`-predicates. Examples of `rel`-predicates are
`owner(user,object)`, which models ownership of objects by users, or
`supervisor(user1,user2)`, which models responsibilities and control within the organizational structure.

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

The authors of [35] present a materialization technique for producing, storing and updating the stable model of the policy. The model is computed on the initial specifications and updated with incremental maintenance strategies.

Note that the clean identification and separation of the four decision stages can be regarded as a basis for a policy specification methodology. In this sense, the choice of a precise ontology

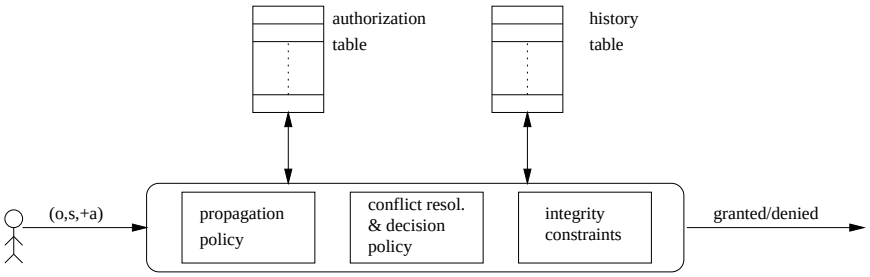


Figure 1.3: Functional authorization architecture in [35]

and other syntactic restrictions (such as those illustrated in Figure 1.2) may assist security managers in formulating their policies.

A general approach to authorization inheritance under the denial-takes-precedence principle can be found in [5]. In this framework, called *hierarchical temporal authorization model* (HTAM), no distinction is made between primitive and derived authorizations. This feature required an extension to the classical stratification techniques.

The syntax of the policy language is the same as the syntax of TABs [4] with one important difference: the elements of authorization triples can be arbitrary nodes of the basic hierarchies. The authorization hierarchy is defined by: $(s, o, a, sgn, g) \leq (s', o', a', sgn, g)$ iff $s \leq s'$, $a \leq a'$ and $o \leq o'$.

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

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

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

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

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

No program containing the above rules is stratifiable, because the two authorizations A^+ and A^- "call" each other through negation-as-failure. This is caused by the negative conditions

not A^+ and not A^- in the bodies of the two rules, that cannot be removed because they are needed to block inheritance when the opposite authorization has a non-defeasible proof (as in the case of explicit exceptions).

Since the above rules are implicit in the semantics of the language, it turns out that the uniqueness of the canonical model of the policy and its PTIME computability cannot be proved by means of the usual stratification techniques. Indeed, the paper extends the theory of logic programming by identifying a class of non-stratifiable programs—called *almost-stratifiable programs*—with the same nice properties as stratifiable programs.

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

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

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

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

A significantly different approach, inspired by *ordered logic programs* [39], can be found in [7]. 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* [39].

More precisely, let a *reference* be a pair $(object, subject)$, and let references be structured as usual, 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 analogous of the *action* field of the authorizations discussed previously), and g is the grantor of the authorization. As in the previous approaches, the semantics is obtained by adapting the stable model semantics.

This syntax is just a factorized reformulation of the syntax of the other approaches. By default, subject and objects are specified by the rule's reference. In rule bodies, one may refer to other subjects and objects by means of referential literals. The real difference between this approach on one hand, and HTAM and FAF on the other hand, is that

when a policy specification has multiple stable models, the authors of [7] propose three different conflict resolution strategies:

1. Use the *well-founded* model of the policy. This (partial) model approximates the intersection of all the stable models of the policy, and can be computed in polynomial time.

2. Use the intersection of the stable models (called the *skeptical semantics* of the policy). Computing the intersection is a coNP-complete problem (data complexity).

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

The second and third strategies are computationally demanding. There exist powerful engines for computing the skeptical and credulous stable semantics [49, 26], but so far they have not been experimentally evaluated in this context. A further difficulty related to the third strategy is that the policy cannot be materialized in advance, because its extension is selected dynamically at access control time.

1.1.3 Message control

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

This approach is pursued in a series of papers by Minsky et al. (e.g., [46, 47]). 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 $\texttt{send}(s, m, t)$, 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 $\texttt{send}(s, m, t)$ onto an action of the form $\texttt{deliver}(m', t')$ 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. Consider the following example:

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

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

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

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

The second paper [47] adapts these ideas to the framework of electronic commerce. Changes mainly concern the set of primitive operations, rule structure is preserved. Moreover, the language distinguishes the act of sending a message from the actual message delivery.

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

Another interesting option is applying a policy description language based on event-condition-action rules, such as $\mathcal{PDL}$ [43, 20], to message handling policies. However, so far $\mathcal{PDL}$ has been considered only in the framework of network management, and static analysis techniques have not been considered as an implementation technique.

1.1.4 Policy composition frameworks

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

Constraint-based approaches

The first approaches to policy composition in the literature are based on constraint languages that express relationships between the elements of different policies. For example, in [30] a constraint may state that object u can/cannot access object v (in this simplified model subjects and objects coincide), where *u and v belong to different sources*. This approach, however, is not based on logic, and focusses mainly on computational complexity results.

Another composition framework [13] deals with *multilevel* databases, and in particular with the problem of merging different, partially ordered sets of *security levels*. In the multilevel approach, each user and each data object is labelled with a security level belonging to a finite set, partially ordered by a relation $\preceq$. To preserve secrecy, a user with security level ℓ is allowed to read only the objects with label $\ell' \preceq \ell$, and write only those with label $\ell'' \succeq \ell$. Composition is formalized following two guiding principles:

1. The interoperation constraints should be satisfied by the composition (trivial as it may seem, this requirement is not formulated explicitly in [30], where some negative constraints may be violated by the composition). The so-called *interoperation constraints* may have the form $\ell_1 : h_1 \preceq \ell_2 : h_2$ or $\ell_1 : h_1 \not\preceq \ell_2 : h_2$. Their intended meaning is that in the merged ordering, the security level ℓ_1 of ordering h_1 must be (resp., must not be) below

the security level ℓ_2 of ordering h_2. The indexes after the colon disambiguate the cases in which the same name has been used with different meanings in different orderings. Conversely, two security levels with different names can be identified with a symmetric pair of constraints, such as $\ell_1 : h_1 \preceq \ell_2 : h_2$ and $\ell_2 : h_2 \preceq \ell_1 : h_1$.

2. The original orderings should not be modified by the composition, that is, for all levels ℓ and ℓ' of any given ordering h, ℓ is below ℓ' in the merged ordering if and only if ℓ is below ℓ' in h. In [30], the equivalent of this requirement is formulated as a pair of properties: the *principle of autonomy* (the permissions granted by an individual source are granted also by the composition) and the *principle of security* (the composition does not introduce any new permissions within any individual source).

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

The results of [13] show that: (i) if the orderings are composable, then the witness is unique up to isomorphism, and (ii) composability can be checked in quadratic time (the algorithm effectively constructs a witness).

If the sources are not composable with respect to the given interoperation constraints IC, then [30, 13] consider the problem of relaxing the constraints to some $IC' \subset IC$ that allows composability. In [30] it is shown that finding such an IC' with maximal cardinality is an NP-complete problem, and this result is extended to the refined framework in [13]. On the contrary, [13] shows that the relaxation problem is in PTIME if IC' is only required to be maximal w.r.t. set inclusion and—possibly—w.r.t. some preference relation over the constraints of IC.

In [13] logic plays two roles. First of all, the constraint language is in fact a logical language. Second, the merging problem is axiomatized with a set of Horn clauses. This is a neat and convenient way of showing some uniqueness and complexity results, by applying the theory of logic programs. Horn clauses can also be used to implement the composability check, but the complexity of this approach is not optimal—it becomes cubic due to the order transitivity axiom that has three variables. The quadratic complexity bound is obtained with a graph-based algorithm.

A simplified version of the constraint system of [13] can be found in [25]. This works is not focussed only on composition and deals with query folding in a mediated framework.

The problem of merging heterogeneous security orderings—and the related technical approach—can be easily adapted to the problem of merging heterogeneous user and object hierarchies (cf. Section 1.1.2). An example can be found in [5].

Deontic approach

Cuppens, Cholvy, Saurel and Carrère [24] tackle the problem of merging security policies expressed in a deontic logic. They focus on conflict detection and resolution.

The given policies may adopt different vocabularies, that can be related to each other by means of first-order formulas. Further axioms formalize the properties of the domain of discourse. These axioms constitute the so-called *domain knowledge* that—in the examples provided in [24]—are essentially Horn clauses with equality.

Technically, when different policies contain conflicting norms, inconsistencies are avoided by indexing the deontic operators with the name of the rules that occur in the given policies. In [24], a rule r is a definite Horn clause whose head is a deontic literal of the form $Op_r\alpha$, where Op_r is an indexed deontic operator and α is a logical atom. Indexing suffices to distinguish the context in which a norm has been formulated. By means of indexed operators, *conflicting rules* can be axiomatized. Two rules r_1 and r_2 are conflicting if there exists a suitable first-order formula s (describing what the authors call a *situation*), such that the union of the domain knowledge with the set of policy rules entails one of the following formulas:

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

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

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

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

- predicates for describing rules, policies and their components as terms,
- predicates for stating that a given policy is more specific than another policy,[3]
- predicates for expressing preferences on policy rules.

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

$$\begin{aligned}\forall r_1, \forall r_2, preferred(S1, r_1, r_2) \leftrightarrow \\ rule(r_1) \wedge rule(r_2) \wedge \\ \exists reg_1, \exists reg_2, \\ regulation(reg_1) \wedge regulation(reg_2) \wedge \\ more\_specific(reg_1, reg_2) \wedge \\ belongs\_to(r_1, reg_1) \wedge belongs\_to(r_2, reg_2)\,.\end{aligned}$$

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

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

Algebraic approach

A more general approach to policy composition is taken in [17], where no assumption is made on the languages used to specify the given policies. The authors note that nonmonotonic policy specification languages can combine different policies, but the result is a *monolithic* specification, which is hard to maintain and verify. The paper provides a list of desiderata for a better policy composition framework:

1. *Heterogeneous policy support* The composition framework should be able to combine policies expressed in arbitrary languages and enforced by different mechanisms. For instance, a data warehouse may collect data from different data sources whose security restrictions may be stated with different specification languages, or refer to different paradigms (e.g., open vs closed policy).

2. *Support of unknown policies* It should be possible to leave part of the policies unknown. It should also be possible to import policies specified and enforced in external systems. These policies are like "black-boxes" for which no (complete) specification is provided. They can be queried at access control time. Think, for example, of a situation where given accesses are subject to a policy P enforcing "central administration approval". While P can respond yes or no to each specific request, neither the description of P, nor the complete set of accesses that it allows might be available. Run-time evaluation is therefore the only possible option for P. In the context of a more complex and complete policy including P as a component, the specification could be partially compiled, postponing at runtime only the evaluation of P (and its possible consequences).

3. *Controlled interference* Policies cannot always be combined by simply merging their specifications, even if they are formulated in the same language. This could have undesired side effects so that the combined policy might not reflect the specifications correctly. As a simple example, consider the combination of two systems P_{closed}, which applies a closed policy, based on rules of the form "*grant access if* $(s, o, +a)$", and P_{open} which applies an open policy, based on rules of the form "*grant access if* $\neg(s, o, -a)$". In the union of the two specifications, the latter decision rule would derive all the authorizations not blocked by P_{open}, regardless of the contents of P_{closed}. Similar problems may arise from uncontrolled interaction of the derivation rules of the two specifications. Moreover, if the adopted language is a logic language with negation, then the merged program might not be stratified (which may lead to ambiguous or undefined semantics).

4. *Expressiveness* The language should be able to express conveniently and in a uniform language a wide range of policy combinations (spanning from minimum privileges to maximum privileges, encompassing priority levels, overriding, confinement, refinement, etc.). The different kinds of combinations must be expressed without changing the input specifications and without ad-hoc extensions to authorizations (like those introduced by some authors to support priorities). For instance, consider a department policy P_1 regulating access to documents and the central administration policy P_2. Assume that access to administrative documents can be granted only if authorized by both P_1 *and* P_2. This requisite can be expressed in existing approaches only by extending explicitly all the rules dealing with administrative documents with the additional conditions specified by P_2. One of the drawbacks of this approach is the resulting rule explosion and the complex

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

5. *Support of different abstraction levels* The composition language should help the identification of the policy components and their interplay at different levels of abstraction. This feature is important to: *i)* facilitate specification analysis and design; *ii)* facilitate cooperative administration and agreement on global policies; *iii)* support incremental specification by refinement.

6. *Formal semantics* The composition language should be declarative, implementation independent, and based on a solid formal framework. An underlying formal framework is needed to: *i)* ensure non-ambiguous behavior and *ii)* reason about policy specifications and prove properties on them [37].

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

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

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

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

1. An *authorization constraint language* $\mathcal{L}_{acon}$ and a semantic relation $\mathsf{satisfy} \subseteq (\mathsf{S} \times \mathsf{O} \times \mathsf{A}) \times \mathcal{L}_{acon}$; the latter specifies for each ground authorization term (s, o, a) and constraint $c \in \mathcal{L}_{acon}$ whether (s, o, a) satisfies c.

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

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

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

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

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

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

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

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

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

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

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

Subtraction (-) The formal semantics is $[\![P_1 - P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \setminus [\![P_2]\!]_e$. Intuitively, the second argument of subtraction specifies exceptions to the authorizations of the firs argument. Subtraction encompasses the functionality of *negative authorizations*. The advantages of subtraction over explicit denials include a simplification of the conflict resolution policies and a clearer semantics, as discussed in [17]. Subtraction can also be used to express different overriding/conflict resolution criteria as needed in each specific context, without affecting the form of the authorizations.

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

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

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

Template (τ) defines a partially specified policy that can be completed by supplying the parameters. $[\![\tau X.P]\!]_e$ is a function over policies (ground authorization sets), such that for all policies S, $[\![\tau X.P]\!]_e(S) \stackrel{\text{def}}{=} [\![P]\!]_{e[X/S]}$. Templates can be instantiated by applying them

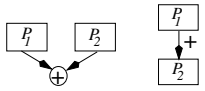
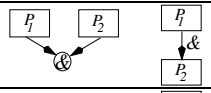
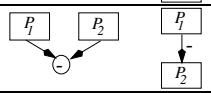
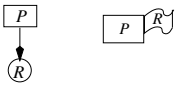
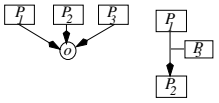
Operator	Semantics $[\![\ ]\!]_e$	Graphical representation
$P_1 + P_2$	$[\![P_1]\!]_e \cup [\![P_2]\!]_e$	
$P_1 \& P_2$	$[\![P_1]\!]_e \cap [\![P_2]\!]_e$	
$P_1 - P_2$	$[\![P_1]\!]_e \setminus [\![P_2]\!]_e$	
$P * R$	$\mathsf{closure}(R, [\![P]\!]_e)$	
$P \hat{} c$	$\{t \in [\![P]\!]_e \mid t \text{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

Figure 1.4: Operators of the algebra and their graphical representation

to a policy expression. For all policy expressions P_1, $[\![(\tau X.P)(P_1)]\!]_e \stackrel{\text{def}}{=} [\![\tau X.P]\!]_e([\![P_1]\!]_e) = [\![P]\!]_{e[X/[\![P_1]\!]_e]}$. We say that all the occurrences of X in an expression $\tau X.P$ are *bound*. The *free* identifiers of a policy expression P are all the identifiers with non-bound occurrences in P. Clearly, $[\![P]\!]_e$ is defined iff all the free identifiers in P are defined in e.

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

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

The expressiveness of the algebra is discussed extensively in [17], by reproducing examples from the literature and by evaluating the algebra w.r.t. the list of desiderata.

Figure 1.4 summarizes the policy composition operators and their semantics. A nice feature of the algebra is that policy specifications can be formulated in a *graphical language*. Figure 1.4 illustrates two possible graphical representations for each operator. Another advantage of an algebraic language is that it is relatively familiar for the users of relational database languages.

Indeed, the fragment of the algebra without closure and templates is a relational algebra over relations with a fixed schema ([17] gives a formal account of this claim).

In [17] it is shown how to translate algebraic specifications into normal logic programs. The reason is that one can then apply well-known *partial evaluation techniques* to materialize compound policies. Intuitively, all the subpolicies that are given a complete specification are reduced to sets of ground authorizations (i.e., their extensions is computed) while the incomplete parts of the specification are simplified to an optimized program with suitable calls to the modules that implement the policy (see [17] for further details).

1.1.5 Trust management

Some logic-based policy languages address data security and privacy protection by means of *trust management* techniques, based on electronic credential negotiation. Trust management encompasses many other issues, so we found it appropriate to devote an entire chapter to this topic. We refer the reader to that chapter for an overview of the existing logic-based policy languages involving credential handling.

1.1.6 XACML

Many of the ideas illustrated so far (including distributed policy composition) have been adopted by XACML, an XML-based standard for policy specification developed by OASIS consortium[4].

The semantics of XACML is not logical; it is expressed in terms of the functional language Haskell. Nonetheless, most of the constructs have a declarative flavor, and could be naturally modelled with a logical semantics. In particular, this is true of conflict resolution and policy combination methods, that are specified procedurally (as algorithms) in XACML while they might very well be expressed declaratively.

XACML specifies not only a policy specification language, but also a policy *query* language. From REWERSE's point of view, the query language is not powerful enough. It assumes the answer may only concern an access control decision: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request can't be answered by this service). Explanations and What-if queries are not supported.

Two more limitations concern trust management and actions (also known as *provisional authorizations* and called *obligations* in XACML), that are mentioned in the specification, but currently not supported by the standard.

As part of REWERSE work, it would be interesting to extend the XACML standard with the aforementioned advanced features, and possibly supply the standard with a logical semantics.

1.2 Policy evaluation and verification

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

[4] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`

Here we are concerned with the problem of identifying potential errors in the policy formulation. In this respect, examples of relevant questions are:

- Is a given policy (or policy module) empty? More generally, we may ask whether each object can be accessed by at least one subject. If this is not the case, then that object cannot be used. These kind of checks has to do with the system's *availability* and is largely independent from the application.

- Similarly, we may ask whether the policy contains all possible authorizations (or, equivalently, no access is ever denied).

- Application-dependent checks may need to decide whether a particular authorization can possibly be granted by the policy, at some time. For example, in order to obey privacy laws, we may ask whether any sensitive piece of data can ever be accessed without the owner's explicit authorization.

Of course, we are interested in static verification tools. Unfortunately, as we already pointed out in Section 1.1.1, the extension of the policy is sometimes undecidable.

For example, Harrison, Ruzzo and Ullman [34] show that it cannot be decided whether a given authorization can become part of a dynamic policy (this is known as the *safety problem*). They prove their undecidability result for a simple imperative policy language supporting user and object creation and deletion.

The other temporal policy languages illustrated in Section 1.1.1 are decidable, and so is the safety problem. However, there is an independent problem affecting both these languages and static languages. The problem is that policy specifications may be partially unknown or unavailable at verification time. For example, the precise set of users and objects may be unknown in advance. Moreover, some policy modules may be either specified later, or be always unavailable as a whole—e.g., because they are formulated by a different organization. Then *the desired properties should hold for all possible ways in which the missing details can be filled in.*

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

This kind of nonmonotonic reasoning had not been tackled before by the literature on nonmonotonic logics. It has been introduced in [15] for the purpose of verifying complex security policies and agent programs. Since all the languages mentioned so far can be embedded into logic programs based on the stable model semantics or variants thereof, the problem of verifying the policies written in these languages can be given a general formulation and general solutions by tackling it in the framework of logic programs.

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

names and object identifiers. The set O identifies the predicates that are not completely defined by the rules in P.

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

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

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

The set of all the completions of $\langle P, F, O \rangle$ is denoted by $\mathsf{Comp}(P, F, O)$. There exist four kinds of *open inference*. They are derived from the two basic forms of inference supported by the stable model semantics: a sentence Ψ is a *credulous* consequence of a logic program P if Ψ holds in *some* stable model of P, while Ψ is a *skeptical* consequence of a logic program P if Ψ holds in *all* the stable models of P.

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

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

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

The four open entailments are pairwise dual.

Proposition 1.2.1 *(Duality) For all open programs Ω and all sentences Ψ,*

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

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

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

Proposition 1.2.2 *(Entailment lattice) Suppose there exists a consistent* $P' \in \mathsf{Comp}(\Omega)$. *Then, for all sentences* Ψ,

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

The problem of computing open inference is still under investigation. In [15] two extensions of resolution are introduced for skeptical open inference and mixed inference of type I, respectively. The goals are conjunctions of implications, and there are 5-6 inference rules. The so-called *failure rule* handles negation-as-failure. It is restricted to predicates $p \notin O$, as expected. The failure rule is expressed in terms of a *counter-support* function that abstracts the actual mechanism for computing negation-as-failure. From the implementation viewpoint, this is the most delicate part of the calculus. In the general setting, a complete implementation of the counter-support function is impossible. Fortunately, the translation of the policy composition algebra into logic programs (cf. Section 1.1) enjoys syntactic restrictions that greatly simplify counter-support computation.

Besides the resolution calculi, in some restricted cases one can apply standard engines such as *DLV* [26] and *Smodels* [49], thanks to the relationships between open programs and answer set programming investigated in [16]. It may also be possible to apply abductive procedures, thanks to the relationships investigated in the same paper.

Bibliography

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

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

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

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

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

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

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

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

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

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

[11] M. Blaze, J. Feigenbaum, Jack Lacy. Decentralized Trust Management. *Proc. of 1996 IEEE Symposium on Security and Privacy*, pp. 164-173, 1996.

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

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

[14] P. Bonatti, P. Samarati. Regulating Service Access and Information Release on the Web, *Proc. of the Seventh ACM Conference on Computer and Communications Security*, 2000. To appear in the Journal of Computer Security.

[15] P.A. Bonatti. Reasoning with open logic programs. Proc. of LPNMR'01, pp. 147-159, LNAI 2173, Springer, 2001.

[16] P.A. Bonatti. Abduction, ASP and Open Logic Programs. Proc. of the International Workshop on Nonmonotonic Reasoning (NMR'02), Toulouse, France, 2002.

[17] P.A. Bonatti, S. De Capitani Di Vimercati, P. Samarati. An Algebra for Composing Access Control Policies, *ACM Transactions on Information and System Security*, 5(1):1-35, 2002.

[18] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proc. Symp. on Security and Privacy*, pages 215–228, Oakland, CA, 1989.

[19] L. Carlucci Aiello, F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. on Computational Logic*, 2(4):542-580, 2001.

[20] J. Chomicki, J. Lobo, S.A. Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. *Proc. of the Seventh International Conference on Principles of Knowledge Representation and Reasoning* (KR 2000), pp. 121-132, Morgan Kaufmann, 2000.

[21] Y-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, M. Strauss. REFEREE: trust management for Web applications. *World Wide Web Journal*, 2(3):706-734, 1997.

[22] F. Cuppens. A logical analysis of authorized and prohibited information flow. *Proc of the IEEE Symposium on Security and Privacy*, pp. 100-109, 1993.

[23] F. Cuppens, C. Saurel. Specifying a security policy: a case study. *Proc. of the Ninth Computer Security Foundations Workshop*, 1996.

[24] F. Cuppens, L. Cholvy, C. Saurel, J. Carrère. Merging Security Policies: Analysis of a Practical Example. *Proc. of the 11th IEEE Computer Security Foundations Workshop* (CSFW 1998), pp. 123-136, IEEE Computer Society, 1998.

[25] S. Dawson, X. Qian, P. Samarati. Providing security and interoperation of heterogeneous systems. *Distributed and parallel Databases*, 8(1):119-145, 2000.

[26] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In J. Dix et al. (eds.), *Proc. of LPNMR'97*, LNAI 1265, Springer Verlag, 1997

[27] T. Fawcett, F.J. Provost. Adaptive Fraud Detection. *Data Mining and Knowledge Discovery*, 1(3):291-316, 1997.

[28] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 1070–1080, MIT Press, 1988.

[29] J. Goguen, J. Meseguer. Unwinding and inference control. *Proc of the IEEE Symposium on Security and Privacy*, 1984.

[30] L. Gong, X. Qian. Computational issues in secure interoperation. *IEEE Trans. on Software Engineering*, 22(1):43-52, 1996.

[31] L. Gong. *Inside Java 2 platform security.* Addison Wesley, 1999.

[32] J.W. Gray, P.F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73-90, 1998.

[33] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 157–166, Washington, D.C., May 1993.

[34] M.A. Harrison, W.L. Ruzzo, J.D. Ullman. Protection in operating systems. *Comm. ACM*, 19(8):461-471, 1976.

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

[36] V. Jones, N. Ching, M. Winslett. Credentials for Privacy and Interoperation. *Proceedings of the New Security Paradigms '95 Workshop*, 1995.

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

[38] W. Lee, S.J. Stolfo, K.W. Mok. A Data Mining Framework for Building Intrusion Detection Models. *Proc. of IEEE Symposium on Security and Privacy*, pp. 120-132, 1999.

[39] N. Leone, P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621-642, 1993.

[40] N. Li, J. Feigenbaum, B. Grosof. A logic-based knowledge representation for authorization with delegation. *Proc. of the 12th IEEE Computer Security Foundations Workshop*, pp. 162-174, 1999.

[41] M. Li, W.H. Winsborough, J.C. Mitchell. Distributed Credential Chaim Discovery in Trust Managament. *Proceedings of 8th ACM Computer and Communication Security*, 2001.

[42] U. Lindqvist, P.A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). *Proc. of IEEE Symposium on Security and Privacy*, pp. 146-161, 1999.

[43] J. Lobo, R. Bhatia, S.A. Naqvi. A Policy Description Language. *Proc. of the of the Sixteenth National Conference on Artificial Intelligence* (AAAI 99), pp. 291-298, AAAI Press, 1999.

[44] J. Lu, G. Moerkotte, J. Schu and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, May 1995.

[45] J. McCarthy. Applications of circumscription in formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[46] N.H. Minsky, D. Rozenshtein. A software development environment for law-governed systems. In P.B. Henderson (ed.), *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (SDE'88), pp. 65-75, 1988.

[47] N.H. Minsky, V. Ungureanu. A Mechanism for Establishing Policies for Electronic Commerce. In *Proc. of the 18th International Conference on Distributed Computing Systems* (ICDCS 1998), pp. 322-331, IEEE Computer Society, 1998.

[48] R.C. Moore. Semantical considerations on non-monotonic logic. *Artificial Intelligence*, 25 (1), (1985).

[49] I. Niemelä, P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In J. Dix, U. Furbach, A. Nerode (eds.), *Logic Programming and Nonmonotonic Reasoning: 4th international conference, LPNMR'97*, LNAI 1265, Springer Verlag, Berlin, 1997.

[50] L. Palopoli, C. Zaniolo. Polynomial-Time Computable Stable Models, *Annals of Mathematics and Artificial Intelligence*, 17(3,4):261-290, 1996.

[51] C.R. Ramakrishnan. A Model Checker for Value-Passing Mu-Calculus Using Logic Programming. *Practical Aspects of Declarative Languages, Third International Symposium* (PADL 2001), LNCS 1990, pp. 1-13, Springer, 2001

[52] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81-132, (1980).

[53] M. Roscheisen, T. Winograd. A Communication Agreement Framework for Access/Action Control. *Proc. of 1996 IEEE Symposium on Security and Privacy*, pp. 154-163, 1996.

[54] D. Saccà, C. Zaniolo. Partial stable models, stable models and non-determinism in logic programs with negation. *Proc. of ACM-PODS*, 1990.

[55] M. Winslett, N. Ching, V. Jones, I. Slepchin. Using Digital Credentials on the World-Wide Web. *Journal of Computer Security*, 5(3), 1997.

[56] K. Sagonas, T. Swift, D.S. Warren, J. Freire, P. Rao. The XSB programmer's manual, Version 2.2. http://xsb.sourceforge.net, 2000

[57] R. Sandhu. Role-based Access Control. *Advances in Computers*, vol. 46, Academic Press, 1998.

[58] K. E. Seamons, W. Winsborough, M. Winslett. Internet Credential Acceptance Policies. *Proceedings of the Workshop on Logic Programming for Internet Applications*, 1997.

[59] G.L. Sicherman, W. de Jonge, R.P. van de Riet. Answering queries without revealing secrets, *ACM Transactions on Database Systems*, 8(1):41-59, 1983.

[60] J.F. Sowa. *Knowledge representation: logical, philosophical, and computational foundations*, Pacific Grove: Brooks/Cole, 2000.

[61] V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu,A. Rajput, T. Rogers, R. Ross, C. Ward. Hermes: Heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes/publications/abstracts/hermes.html.

[62] 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.

[63] T. Yu, M. Winslett, K.E. Seamons. Interoperable Strategies in Automated trust Negotiation. *Proceedings of 8th ACM Computer and Communication Security*, 2001.

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

[65] K.R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, San Mateo, 1988.

[66] C. Baral and V.S. Subrahmanian. Stable and extension class theory for logic programs and default theories. *Journal of Automated Reasoning*, 8:345–366, 1992.

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

[68] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996.

[69] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.

[70] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.

[71] J. Chomicki, J. Lobo and S. Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management In *KR2000: Principles of Knowledge Representation and Reasoning* pages 121–132, San Francisco, CA, USA, 2000.

[72] Y-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[73] G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2(3):397–425, 1992.

[74] J. Schlipf K.A. Berman and J.V. Franco. Computing the well-founded semantics faster. In A. Nerode W. Marek and M. Truszczynski, editors, *Proc. 1995 Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, pages 113–126, 1995.

[75] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[76] N. Li, B.N. grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 27–42, Oakland, CA, 2000.

[77] T. F. Lunt. Access control policies for database systems. In C. E. Landwehr, editor, *Database Security II: Status and Prospects*, pages 41–52. North-Holland, Amsterdam, 1989.

[78] F. Mayer M. Branstad, H. Tajalli and D. Dalva. Access mediation in a message passing kernel. In *Proc. IEEE Symp. on Security and Privacy*, pages 66–72, Oakland, CA, 1989.

[79] W. Marek and V.S. Subrahmanian. The relationship between stable, supported, default and auto-epistemic semantics for general logic programs. *Theoretical Computer Science*, 103:365–386, 1992.

[80] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Prog. Lang. and Systems*, 4(2):258–282, 1982.

[81] N.H. Minsky and D. Rozenshtein. A software development environment for law-governed systems In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical software development environments*, Boston, MA USA, 1988.

[82] N.H. Minsky and V. Ungureanu. A Mechanism for Establishing Policies for Electronic Commerce In *International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, 26-29 May, 1998.

[83] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases*, pages 193–216. Morgan Kaufmann, San Mateo, 1988.

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

[85] G. Martella S. Castano, M.G. Fugini and P. Samarati. *Database Security*. Addison-Wesley, 1995.

[86] P. Samarati S. Jajodia and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proc. IEEE Symp. on Security and Privacy*, pages 94–107, Oakland, CA, 1997.

[87] K. E. Seamons, W. Winsborough, and M. Winslett. Internet credential acceptance policies. In *Proceedings of the Workshop on Logic Programming for Internet Applications*, Leuven, Belgium, July 1997.

[88] V.S. Subrahmanian S. Jajodia, P. Samarati and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. of the 1997 ACM Internationa SIGMOD Conference on Management of Data*, Tucson, AZ, 1997.

[89] O. S. Saydjari, S. J. Turner, D. E. Peele, J. F. Farrell, P. A. Loscocco, W. Kutz, and G. L. Bock. Synergy: A distributed, microkernel-based security architecture, version 1.0. Technical report, National Security Agency, Ft. George G. Meade, MD, 1993.

[90] H. Shen and P. Dewan. Access control for collaborative environments. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, pages 51–58, 1992.

[91] A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. 8-th ACM Symposium on Principles of Database Systems*, pages 1–10, Philadelphia, 1989.

[92] W. Winsborough, K. E. Seamons, and V. Jones. Automated trust negotiation. In *Proc. of the DARPA Information Survivability Conf. & Exposition*, Hilton Head Island, SC, USA, January 25-27 2000. IEEE-CS.

[93] M. Winslett, N. Ching, V. Jones, and I. Slepchin. Assuring security and privacy for digital library transactions on the web: Client and server security policies. In *Proceedings of ADL '97 — Forum on Research and Tech. Advances in Digital Libraries*, Washington, DC, May 1997.

[94] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, December 1994.

[95] T. Yu, X. Ma, and M. Winslett. An efficient complete strategy for automated trust negotiation over the internet. In *Proceedings of 7th ACM Computer and Communication Security*, Athens, Greece, November 2000.

Chapter 2

Trust Management

2.1 Introduction and motivation

The important role of trust and trust management in the development of modern open distributed and decentralized systems has been recognized by many researchers. Trust has been studied in the context of decentralized access control, e.g. [11, 32], public key certification, e.g. [7, 14], reputation systems for P2P networks, e.g. [2, 30, 50], and mobile ad-hoc networks, e.g. [41].

The problem of trust has been also identified as one of the major difficulties for the Semantic Web [20, 43]. Focused on issues pertaining to knowledge representation and ontology design, the Semantic Web is emerging as a large and uncensored system to which anyone can contribute and which anyone can access. However, this raises questions with regard to the trustworthiness of the provided services and of the users accessing them. Thus, the issue of trust management as solution to establishing and maintaining trust relationships among participating parties is a major aspect for the success of the Semantic Web.

This chapter investigates the state of the art in the area of trust and trust management in open distributed and decentralized systems.

The structure of the chapter is as follows. We start with defining the concept of trust, investigate its components and properties, and give a taxonomy of different classes of trust. Then we define the concept of trust management and distinguish the two main approaches for trust management - the policy-based and the reputation-based. Following, sections 2.2 and 2.3 survey these two approaches.

2.1.1 Definitions, examples, and taxonomy of trust

Trust is a key issue in any interaction process, and humans apply trust constantly, every day. Thus, trust has been the subject of studies for many disciplines such as sociology, psychology, economy, as well as, more recently, computer science. However, one common feature of these studies is the lack of a clear and generally agreed definition of the trust concept itself [44].

A variety of trust definitions have been put forward and investigated in [44, 21]. For the purpose of our survey we adopt the following definition of trust by Grandison [21]:

Trust is "the firm belief in the competence of an entity to act dependably, securely, and reliably within a specified context".

Trust components and properties

Despite the variety of existing definitions, researchers tend to concur in modelling trust as a relationship between two parties: the *trustor*, the entity which trusts, and the *trustee*, the entity which is trusted by the trustor.

In general, trust is not symmetric. That is, if Alice trusts Bob, this does not imply that Bob should trust Alice. Trust is a directed relation from Alice to Bob, it is the belief of Alice regarding Bob. A possible mutual trust between Alice and Bob is modelled using two directed trust relations.

Trust is usually seen as having a purpose or a *context*. For instance, Alice trusts Bob as a doctor, but she might not trust Bob as a car mechanic.

In addition, a trust relation might also bear a *trust level*, which can be quantitative or qualitative, characterizing the degree to which the trustor trusts the trustee. For instance, Alice might trust Bob as a doctor very much, while she only moderately trusts Martin as a doctor. Computing the right trust level is a major concern in reputation-based trust management.

Grandison [21] and Abdul-Rahman [1] discus a number of properties of trust which are important in the context of electronic transactions in distributed systems:

- Trust is subjective. Different observers may have different perception of the same entity's trustworthiness.
- Trust is transitive, e.g. in [11], or non-transitive, e.g. in [1]. If transitive, when Alice trusts Bob and Bob trusts Anna, Alice will trust Anna.
- Trust is dynamic. That is, trust is continuously changing over time.
- Trust is not monotonic. Further observations may elevate or lower the level of trust that is invested in another entity.

The question about the transitivity of trust is a point of disagreement. For instance, starting from the sociological characteristics of trust, Abdul-Rahman and Hailes [1] suggest that trust is not inherently transitive. This opinion has been shared by [15], which formally argues that trust transitivity could lead to *unintentional transitivity*. In such a situation, when Alice trust (transitively) Bob, Bob can add to the trust assertions of Alice, without hers explicit consent.

On the other hand, trust transitivity has been widely accepted as a mean for decentralization in policy-based mechanisms and also as tool for increasing the quantity of reached recommendations in a reputation system. To cope with the problems of unintentional transitivity, approaches such as limiting delegation depth [31] or trust decaying along recommendation paths have been adopted [14].

Taxonomy of trust

Grandison [21] identifies a number of trust classes. As stated by the author, the taxonomy might not be exhaustive. The identified trust classes are: access to trustor's resources, provision of trust by the trustor, certification of trustees, delegation, and infrastructure trust.

Trust has been also segregated on two other dimensions. First, distinction has been made between trust in an entity to perform an action, and trust in an entity to recommend other entities to perform the action. This is the distinction between Alice trusting Bob as a doctor, and Alice trusting Bob to recommend a good doctor. Second, following the existence of

recommenders, distinction has been made between trust resulting from direct observation and assessment of the trustee and trust that is derived from the trust conveyed by the recommenders. This is the distinction between Alice trusting Bob as doctor, trust resulting from Alice's own experience with Bob, and Alice trusting Bob as a doctor, based on the fact that she trusts Bill as a recommender for a good doctor and on the fact that Bill trusts (and recommends) Bob to be a good doctor.

2.1.2 Trust management

In their seminal paper [11], Matt Blaze and colleagues have coined the term *trust management* as "a unified approach to specifying and interpreting security policies, credentials, and relationships which allow direct authorization of security-critical actions". However, as shown in [21] this definition is limiting and focused on authorization. The following broader definition has been proposed instead:

Trust management is "the activity of collecting, encoding, analyzing and presenting evidence relating to competence, honesty, security or dependability with the purpose of making assessments and decisions regarding trust relationships"

Two main approaches are currently available for managing trust:

Policy-based trust management This approach has been proposed in the context of open and distributed services architectures as well as in the context of Grids systems as a solution to the problem of authorization and access control in open systems. The focus here is on trust management mechanisms employing different policy languages and engines for specifying and reasoning on rules for trust establishment. The goal is to determine whether or not a ceratin priori unknown user can be trusted, based on a set of credentials and a set of policies.

Reputation-based trust management This approach has emerged in the context of electronic commerce systems, e.g. eBay. In distributed settings, reputation-based approaches have been proposed for managing trust in public key certificates, in P2P systems, mobile ad-hoc networks, and, very recently, in the Semantic Web. The focus here is on trust computation models capable to estimate the degree of trust that can be invested in a certain party based on the history of its past behavior.

In the following sections we survey the two trust management approaches.

2.2 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 attributes. 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. This section focuses on access control and describes in detail the state of the art of policy-based trust management.

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.

Currently, 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. Therefore, 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 the fact of him or her being a student or not. These mechanisms are property-based and, in contrary to identity-based systems, provide the scalability necessary nowadays.

2.2.1 Trust Management

Existing authorization mechanisms were not enough to provide powerful and robustness for handling security in a scalable manner as it is required in the current World Wide Web. For example, Access Control Lists (ACL) are lists describing the access rights a principal (entity) has on an object (resource). An example is the file system permissions mechanism used in the UNIX operating system. However, although ACLs are easy to understand and they have been used extensively, they lack of the following:

- Authentication: ACL requires that entities are known in advance. This assumption might not hold in true distributed environments where an authentication pre-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)
- Expressibility and Extensibility: A generic security mechanism must be able to be extended 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* [11] 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 [52] or X.509 [25]) are:

1. Obtain the certificates, verify the signatures and determine public keys of issuers.
2. Check if the certificates have been revoked.
3. Search for a trust chain between the certificate's public key and a trusted entity.
4. Extract names from certificates
5. Map names into actions that they are allowed to perform.
6. Check if the requester is authorized to perform the requested action according to the local policy.

7. Accept the request if everything was valid.

This can be summarize as "is the key, with which the request was signed, authorized to perform the requested action?". However, some of these steps are too specific and can be generalized integrating policy specifications with the binding of public keys to authorized actions. The previous steps would be reduced to:

1. Obtain the certificates, verify the signatures and determine public keys of issuers.
2. Check if the certificates have been revoked.
3. Use a local "trust management engine" with the request, certificates, and descriptions of the local policy as input.
4. Proceed if the request was approved.

or what is the same, "given a set of credentials, do they prove that the requested action complies with a local policy?". In [11, 9] 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.

In the next sections, some of the systems are described that try to provide a scalable framework following these guidelines.

PolicyMaker

PolicyMaker [11, 12] 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-like certificates (PGP [52] and X.509 [25]) 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 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. The syntax of assertions is:

Source **ASSERTS** *AuthorityStruct* **WHERE** *Filter*

and can be read as *Source* trusts the public keys enumerated in *AuthorityStruct* to be associated with action strings that satisfy *Filter*. Therefore, a *Source* represents the source of the assertion. There are two types of assertions depending on whether the *Source* is the local

policy (policy assertions) or the public key of a third entity (signed assertions or certificates). *AuthorityStruct* specifies the public key to whom the assertion applies. Action strings must satisfy the predicate in *Filter* for the assertion to hold.

The PolicyMaker interpreter receives queries of the form

$key_1, key_2, ..., key_n$ **REQUESTS** *ActionString*

A public key or a sequence of public keys request an action string. Action strings depend on the application and PolicyMaker does not even need to know their semantics.

The system may run in two different modes. The first one simply returns if the query is satisfied or not (accepts or rejects action strings). The second one might add some annotations to an accepted action string indicating restrictions or extra information.

REFEREE

REFEREE [16] (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. There are two approaches to eliminate potential security problems. The first one is to eliminate dangers (like for example Java applets which are executed in an environment where only harmless actions can be performed). The second one is to apply trust. The definition given in [16] is *To trust is to undertake 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 [42] 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. Therefore, 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.

REFEREE improves PolicyMaker in the sense that PolicyMaker [11, 12] 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.

In REFEREE there are three kind of data types:

- Tri-values: is one of true, false or unknown.
- Statements lists: is a collection of assertions. A statement is formed by some content and a context for the content. Content and context are s-expressions. The interpretation of the context depends on the agreement between REFEREE and the calling application. A statement list is an unordered list of statements.
- Programs: A program can be a policy or a credential.

A program takes a statement list defining the current evaluation context and required/optional extra arguments as an input. It returns a tri-value (the result of the program) and a statement list (a justification). The program returns true if it was possible to infer compliance with a policy (credentials were sufficient to grant the requested action). It returns false

if it was not possible to infer compliance (credentials were sufficient not to grant the action) or unknown if no inference could be made at all (credentials are not sufficient to take a decision: neither for approving nor denying the action).

Profiles-0.92 Language *Profiles-0.92* is the language used in REFEREE and it has the following features:

- Appending of statements returned by invoked programs
- The Load-labels invocable program
- Tri-value Combinators and Operators. The boolean operators AND, OR and NOT are extended to manage tri-values. Therefore, for example, *AND true unknown* is evaluated to *unknown*. The operators *true-if-unknown* and *false-if-unknown* translate the tri-values into boolean values.
- Statement-list pattern matching

In addition, the keyword *invoke* is used to call another REFEREE program.

KeyNote

KeyNote [8, 10] 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.

In KeyNote, policies and credentials are, in general, called assertions. They both are specified using the same format. The main difference between them is that policies are locally trusted (and therefore they do not need any signature). An example of a KeyNote assertion extracted from [9] is depicted in figure 2.1.

Programs in KeyNote are specified in the *Conditions* field. In *Licensees* the principal or principals are specified to which authority is delegated. In order to satisfy an assertion, both the *Conditions* and the *Licensees* fields must be satisfied.

A picture with the architecture of the KeyNote[1] system is depicted in figure 2.2.

As well as PolicyMaker, KeyNote does not enforce policies but gives advise to applications that call it. It is up to the calling application whether to follow KeyNote's advises or not.

[1]The figure has been extracted from http://www.crypto.com/trustmgt/kn.html

```
KeyNote-Version: 1
Authorizer: rsa-pkcs1-hex:"1023abcd"
Licensees: dsa-hex:"86512a1" ||
           rsa-pkcs1-hex:"19abcd02"
Comment: Authorizer delegates read
           access to either o the
           Licensees
Conditions: ($file == "/etc/passwd" &&
           $access == "read") - >
               { return "ok" }
Signature: rsa-md5-pkcs1-hex:"f00f5673"
```

Figure 2.1: Sample KeyNote assertion

SD3

SD3 [26] (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 [17]. 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".

Figure 2.3 shows the structure of the evaluator. It consists of three elements: an optimizer, a cache, and a core evaluator. The more novel implementation techniques of the evaluator as well as the theoretical foundations of SD3 are described in [47].

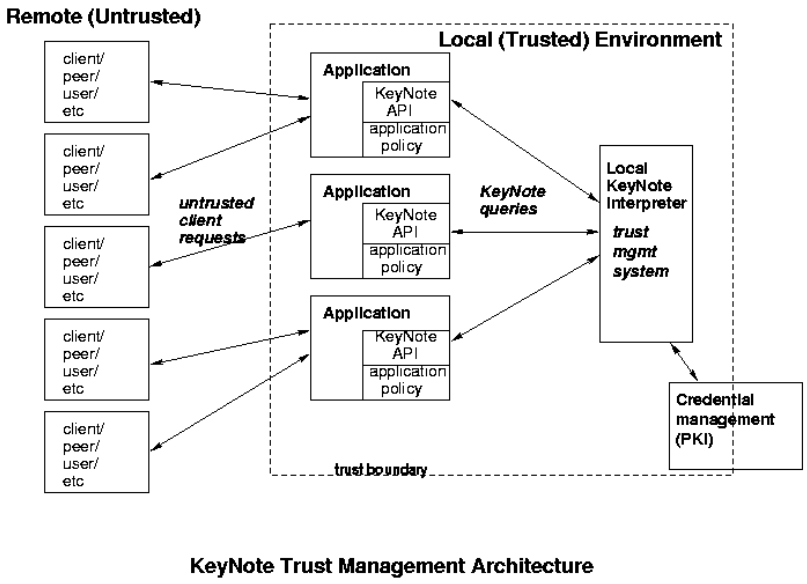


Figure 2.2: The KeyNote Architecture

2.2.2 Trust Negotiation

In traditional distributed environments, service providers and requesters are usually known to each other. Often, shared information in the environment tells which parties can provide what kind of services and which parties are entitled to make use of those services. Thus, 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. In this case, 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, the Semantic Web provides an environment where 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:

1. Trust between two strangers is established based on parties' properties, which are proved

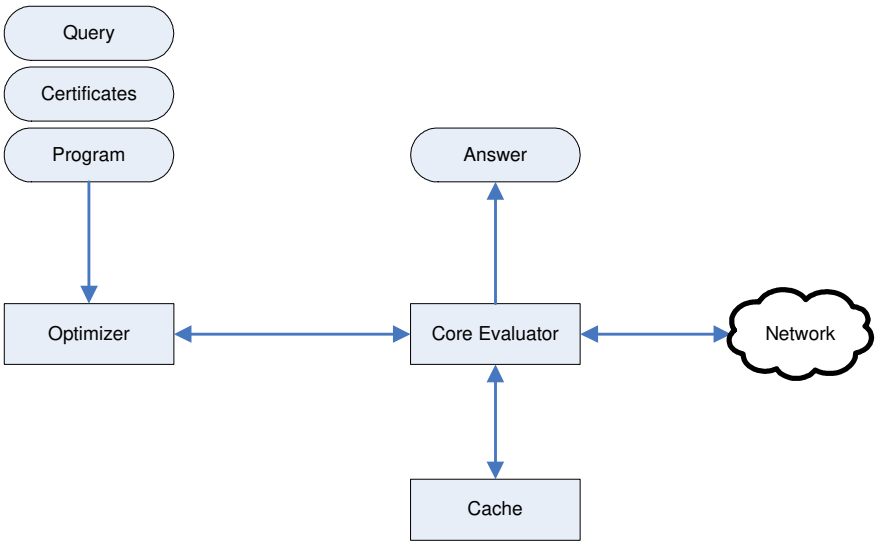


Figure 2.3: SD3 Evaluator

through disclosure of digital credentials.

2. 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.

3. 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. (For uniformity of terminology, we will say that R is *disclosed* when "Peer1" grants "Peer2" access to R.)

In practice, trust negotiation is conducted by security agents who interact with each other on behalf of users. A user only needs to specify policies for credentials and other resources. The actual trust negotiation process is fully automated and transparent to users. Further, the above example used objective criteria for determining whether to allow the requested access. More subjective criteria, such as ratings from a local or remote reputation monitoring service, can also be included in a policy.

Before we delve into details, though, let us highlight two general criteria for trust negotiation languages as well as two important features already mentioned briefly above. A more detailed

discussion can be found in [45].

Well-defined semantics Two parties must be able to agree on whether a particular set of credentials in a particular environment satisfies a policy. To enable this agreement, a policy language needs a clear, well-understood semantics.

Expression of complex conditions A policy language for use in trust negotiation needs the expressive power of a simple query language, such as relational algebra plus transitive closure. Such a language allows one to restrict attribute values (e.g., age must be over 21) and relate values occurring in different credentials (e.g., the issuer of the student ID must be a university that ABET has accredited).

Sensitive policies The information in a policy can reveal a lot about the resource that it protects. For example, who is allowed to see Alice's medical record—her parole officer? Her psychiatrist or social worker? Because policies can contain sensitive information, and because they may be shown to outsiders, they need to be protected like any other shared resource.

Delegation Trust negotiation research has also addressed the issue of delegation of authority. For example, rather than issuing student IDs directly, a university may delegate that authority to its registrar. Then student IDs from that university will not bear the digital signature of the university itself, but rather the signature of the registrar.

RT: Role-based Trust-Management

The RT framework [35, 33, 34] 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: entities must be able to assert that other entity has an attribute.
- Delegation of attribute authority: an entity can delegate the authority over an attribute to a different entity.
- Inference of attributes: attributes can be used to infer about other attributes.
- Attribute field: attribute credentials could also contain field values (e.g. age). They can be used to infer other attributes (e.g. age > 21).
- Attribute-based delegation of attribute authority: it is possible to delegate on entities which are only known and trust based on certified attributes.

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 parts which are now described.

RT_0 RT_0 [35] 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. The following describe the different kind of construtions allowed in RT_0:

- Simple member ($K_A.R \leftarrow K_D$)
  The principal K_D is a member off the role $K_A.R$.

- Simple containment ($K_A.R \leftarrow K_B.R_1$)
  The role $K_A.R$ contains any principal that is a member of the role $K_B.R_1$.

- Linking containment ($K_A.R \leftarrow K_A.R_1.R_2$)
  The role $K_A.R$ contains every role of the form $K_B.R_2$ for each K_B which is a member of the role $K_A.R_1$.

- Intersection containment ($K_A.R \leftarrow K_{B_1}.R_1 \cap \ldots \cap K_{B_i}.R_i$)
  The role $K_A.R$ contains the intersection of the members of the roles $K_{B_1}.R_1 \cap \ldots \cap K_{B_i}.R_i$.

- Simple delegation ($K_A.R \Leftarrow K_B : K_C.R_2$)
  In this statement, K_A delegates its control over R to K_B. If $K_C.R_2$ is present, K_A restricts its delegation in such a way that K_B can only assigned members of $K_C.R_2$ to be members of $K_A.R$.

- Linking delegation ($K_A.R \Leftarrow K_A.R_1 : K_C.R_2$)
  K_A delegates control over R to all the members of $K_A.R_1$ and the delegation is controlled so only members of $K_C.R_2$ can be assigned as members of $K_A.R$.

RT_1 In RT_0 roles do not take any paremeters. RT_1 role definitions have the same form than the one 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 constraint 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. Manifold roles are constructed as follows:

- Product containment ($K_A.R \leftarrow K_{B_1}.R_1 \odot \ldots \odot K_{B_k}.R_k$)
  The role $K_A.R$ contains every principal set p such $p = p_1 \cup \ldots \cup p_k | p_i is a member of k_{B_i}.R_i$.

- Exclusive product containment ($K_A.R \leftarrow K_{B_1}.R_1 \otimes \ldots \otimes K_{B_k}.R_k$)
  The role $K_A.R$ contains every principal set p such $p = p_1 \cup \ldots \cup p_k | p_i \cup p_j = \phi for 1 < p_i \neq p_j < k and p_i is a member of k_{B_i}.R_i$.

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 $D \stackrel{D\ as\ A.R}{\longleftarrow} B_0$. 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 $B_0 \stackrel{D\ as\ A.R}{\longleftarrow} B_1$.

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 [13]. 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)*
- *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.

Service accessibility rules are subdivided in

- $service\_prereqs(s(L))$: service prerequisite rules define required credentials and declarations which are a necessary condition for service access
- $service\_reqs(s(L))$: service requisite rules define required credentials and declarations which are a sufficient condition for service access
- $facet\_reqs(s(L), f)$: facet requisite rules define required credentials and declarations necessary to apply a facet to a service.

Portfolio requisite rules ($release\_reqs(o)$) 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.

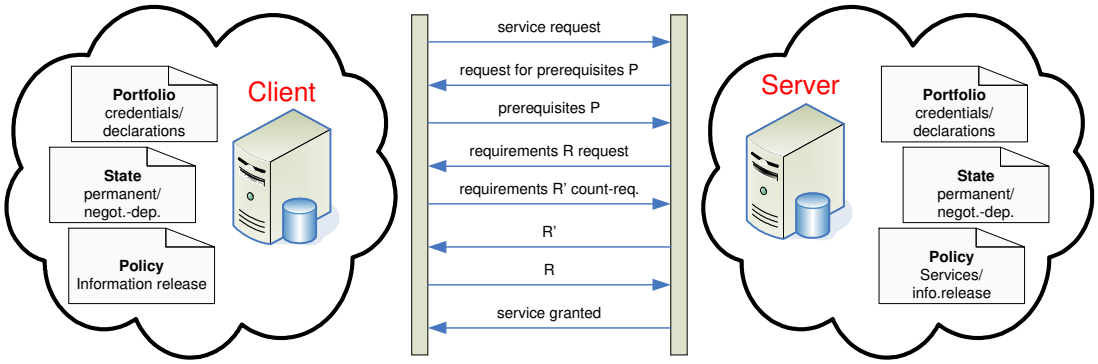


Figure 2.4: Client/Server interplay

In figure 2.4 is shown an example scenario of the interaction process between client and server.

PeerTrust

PeerTrust [19, 4, 38, 39] builds upon the previous work on policy-based access control and release for the Semantic Web by showing how to use *automated trust negotiation*.

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 [36], which have been used as the basis for the rule layer of the Semantic Web and specified in the RuleML effort ([22, 23]) as well as in the recent OWL Rules Draft [24]. 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, we will instead focus on 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, we extend each literal lit_i 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. We accomplish this with *Context* literals that represent release policies for literals and rules, so that we now have literals and rules of the form

lit_i @ Authority \$ $context_j$
$lit_i \leftarrow_{context_j} lit_1, \ldots, lit_{i-1}$

For example, suppose that "Peer1" has derived a clause C and it wishes to send this literal to "Peer2". It can only do so if it is able to derive C *\$ Requester = "Peer2"*. Here, Requester is a pseudovariable whose value is automatically set to the party that "Peer1" is trying to send the literal or rule. If no context is specified for a literal or a rule, the default context 'Requester = Self' applies, implying that the literal or rule cannot be sent to any other peer. 'Self' is a pseudovariable whose value is a distinguished name of the local peer. The release policy for a literal can be cleanly specified in rules separate from those used to derive the literal, e.g.,

$p(X_1, \ldots, X_n)$ \$ $context_p$ $(X_1, \ldots, X_n$, Requester, Self) $\leftarrow p(X_1, \ldots, X_n)$

In this document, we will strip the contexts from literals and rules when they are sent to another peer. However, sticky policies can be implemented by leaving contexts attached to literals and rules in messages and defining how to propagate contexts across modus ponens, so that a peer can control further dissemination of its released information in a non-adversarial environment.

Using the *Authority* and *Context* arguments, we can delegate evaluation of literals to other peers and also express interactions and the corresponding negotiation

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. We assume 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.

Implementation PeerTrust 1.0's outer layer is a signed Java application or applet program, which keeps queues of propositions that are in the process of being proved, parses incoming queries, translates them to the PeerTrust language, and passes them to the inner layer. Its inner layer answers queries by reasoning about PeerTrust policy rules and certificates using

Prolog metainterpreters (in MINERVA and XSB Prolog, whose Java implementation offers excellent portability), and returns the answers to the outer layer. PeerTrust 1.0 imports RDF metadata to represent policies for access to resources, and uses X.509 certificates and the Java Cryptography Architecture for signatures. It employs secure socket connections between negotiating parties, and its facilities for communication and access to security related libraries are in Java. Figure 2.5 shows current PeerTrust architecture.

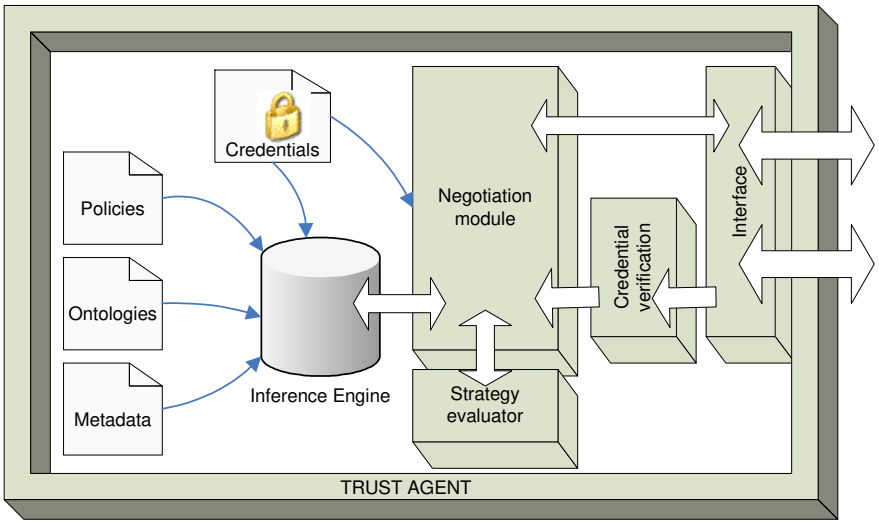


Figure 2.5: Peertrust Architecture

In figure 2.6 is depicted an implemented scenario in an e-learning domain. 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 [5, 6] 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

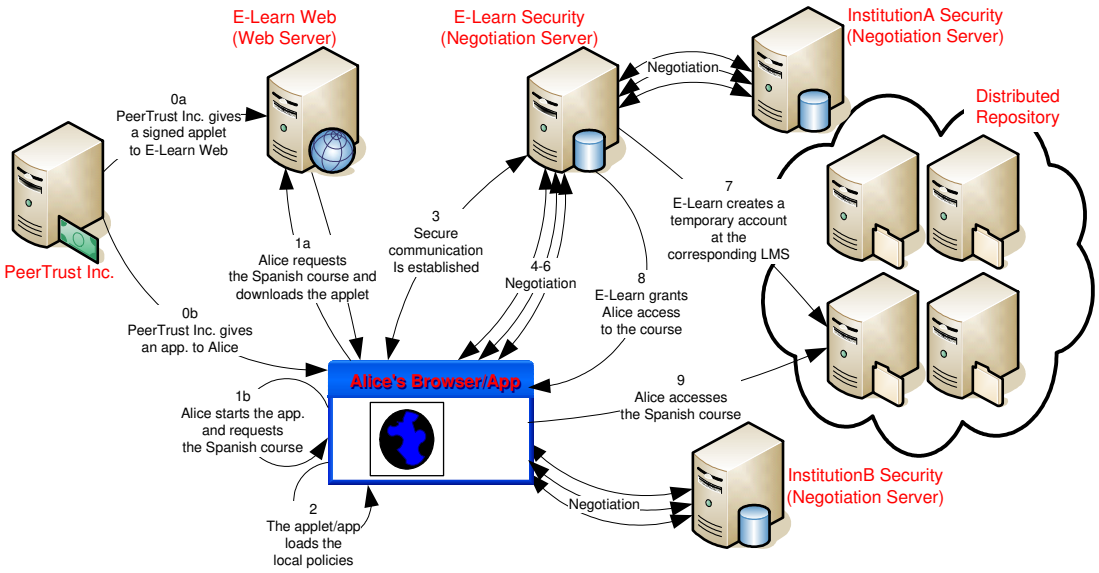


Figure 2.6: Peertrust: Automated Trust Negotiation for Peers on the Semantic Web

- $canDeactivate(e, r)$ specifies who can revoke which role
- $isDeactivated(e, r)$ is used to define automatically triggered role revocation
- $canReqCred(e_1, e_2, p(e))$ specifys the requirements that a request must satisfy in order to issue and disclose credentials

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 to extend 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}.
Table 2.1 shows a summary of the syntax of the Cassandra policy language.

Predicate names
p ::= canActivate | hasActivated | permits | canDeactivate |
isDeactivated | canReqCred, and user-defined predicate names
Policy rule
$E_{loc}@E_{iss}.p_0(e_0) \leftarrow loc_1@iss_1.p_1(e_1), \ldots, loc_n@iss_n.p_n(e_n), c$
Credential(rule)
$E_{loc}@E_{iss}.p_0(e_0) \leftarrow c$
Aggregation rule
$E_{loc}@E_{loc}.p(agg-opx, y) \leftarrow E_{loc}@iss.q(x), c$
where agg-op is group or count
C_{eq} expressions
e ::= x | E
C_{eq} constraints
c ::= true | false | e = e | c $\wedge$ c | c $\vee$ c
C_0 expressions
e ::= x | E | N | C | () | $(e_1, \ldots, e_n)$ | $\pi_i^n(e)$ | $R(e_1, \ldots, e_n)$ | $A(e_1, \ldots, e_n)$ |
$f(e_1, \ldots, e_n)$ | ϕ | Ω | $e_1, \ldots, e_n$ | $e - e$ | $e \cap e$ | $e \cup e$
C_0 constraints
c ::= true | false | $e = e$ | $e < e$ | $e \subseteq e$ | $c \wedge c$ | $c \vee c$
and derivable constraints c ::= $\ldots$ | $e \in e$ | $e \notin e$ | $e \in [e_1, e_2]$ | $[e_1, e_2] \subseteq [e_1, e_2]$
C_0 types
τ ::= entity | int | const | unit | $\tau_1 \times \ldots \times \tau_n$ | role(τ) | action(τ) | set(τ)
Access-control operations
doAction($A(e)$), activate($R(e)$), deactivate($E_v, R(e)$), reqCred($E_s@E_{iss}.p(x) \leftarrow c$)

Table 2.1: Cassandra policy language syntax

2.3 Reputation-based trust management

In a distributed and decentralized system each peer builds trust in the peers she has interacted with based on the history of their encounters. However, building trust only based on individual experiences is limiting because of at least two reasons. First, it does not enable peers to trust other peers they have not encountered yet, potentially missing good opportunities for interactions. Second, it does not take in consideration the potential huge amount of others' experiences with same peers, affecting the accuracy of trust evaluation.

To address this problem researchers have exploited mechanisms pertaining to the social trust networks formation. That is, peers would base their trust not only on their individual experiences but also on the recommendations received from others. This word of mouth mechanism has been successfully implemented in centralized recommender systems for e-commerce such as eBay and Amazon. In decentralized settings, the word of mouth principle has been proposed for managing trust in public key certificates (e.g. PGP), peer-to-peer (P2P) systems, and, very recently, in the Semantic Web.

A major problem in these systems is the computation of trust from experiences and received recommendations. The questions are: a) how to aggregate the individual experiences into trust, b) how to aggregate the trust along a recommendation path, and c) how to aggregate the trust across possible multiple recommendations paths.

The existing systems can be distinguished based on their approaches to the above questions.

However, they can be also split into two major categories based on whether they consider trust to be transitive or not.

With regard to trust transitivity, most of the reputation systems in P2P consider (implicitly or explicitly) trust to be non-transitive. They focus on aggregating experiences and recommendations into trust, while also dealing with malicious recommenders.

On the other hand, the computational models from public key certification and the Semantic Web consider trust to be transitive. The algorithms proposed there are capable of "walking the web of trust" [14] and compute the trust based on the recommendations of a friend of friend, etc.

Following, we investigate the existing problems and solutions related to trust computational models both in P2P reputation systems as well as in the web of trust.

2.3.1 Trust computation in P2P reputation systems

In P2P the global interaction history is distributed over all peers and it is not available to a single peer. Trust based on the global history is often just a theoretical notion and it is called *global trust*. A single peer has a subset of the global transaction history when determining trust. We call that type of trust *local trust* or just *trust*. In particular, if trust is based only on own experience (not recommendations) then we call it *direct trust*.

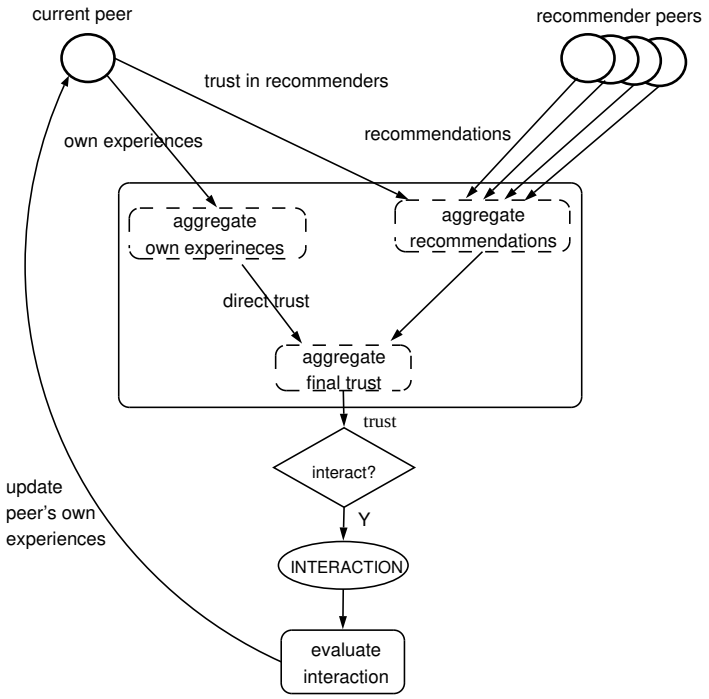


Figure 2.7: The model of computing local trust

Figure 2.7 depicts a generalization of the trust computation model as proposed by Abdul-Rahman and Hails [1]. The model assumes the following scenario.

Peer Alice searches the P2P network for a certain service. She finds that the service is provided by peer Bob. Alice would interact with Bob, but she does not know whether Bob is trustworthy or not. Therefore, in order to learn the reputation of Bob, Alice asks other peers for their recommendations regarding Bob.

Following Alice's request, a number of peers respond with recommendations. However, some of the recommenders might be malicious. Thus, when evaluating the reputation, Alice takes into account her previous experience with the recommenders, in order to determine their trustworthiness.

Once Bob's reputation is computed Alice inspects also her own experience history to find records of previous interactions with Bob. That experience is then combined with previously computed reputation to determine the (local) trust Alice can invest in Bob.

If the computed trust exceeds a certain trust threshold required for the given context, then Alice will interact with Bob.

After the interaction, Alice evaluates the interaction's outcome and updates her experiences with Bob. Also, Alice evaluates the recommendations she received for Bob and updates her trust in the recommenders.

Based on this model we outline the major approaches for computing trust in P2P reputation systems.

Notation

P is the set of peers. D_t is the domain of the trust function, denoting the level or the degree of trust. The domain of trust varies from system to system. For instance in a probabilistic approach D_t is the interval $[0, 1]$.

Trust is context dependent. However, for the sake of simplicity we consider that all the trust computation is about one context only.

The level of trust peer a invests in peer b is the function $t(a, b)$, defined as:

$$t : P \times P \mapsto D_t$$

We can also define a global trust function, which shows the reputation of a ceratin peer based on all the experiences with that peer:

$$gt : P \mapsto D_t$$

The direct trust is the trust which result solely from peers own's experiences:

$$dt : P \times P \mapsto D_t$$

The trust $t(a, b)$ is the result of combining the direct trust $dt(a, b)$ and the recommendations received from others.

The recommendation a gives for b will be denoted by $rec(a, b)$ and defined as:

$$rec : P \times P \mapsto D_t$$

Similarly, the recommender trust, that is a peer's belief that another peer is trustworthy for giving recommendations about other peers, is given by:

$$rt : P \times P \mapsto D_t$$

As already stated, when computing trust one can take own experience and/or recommendations given by other peers.

The direct trust depends on the history of previous transactions. Therefore, every peer has to keep a record of experiences from previous interactions. The set of experiences of a peer a is denoted by Q_a where $Q_a \subseteq a \times P \times D_s \times Time$. The tuple $(a, b, s, t) \in Q_a$ denotes a "record" with the following meaning: peer a has interacted with b at the time point t and the a's is satisfaction degree is s. The satisfaction values range over the domain D_s. The D_s is the same as D_t, but need not to be. The global set of experiences of all peers can also be defined as $Q \subseteq P \times P \times D_s \times Time$.

The set $W(b) \subseteq P$ denotes a set of peer witnesses who have interacted with b and can provide recommendations to the others with regard to b:

$$W(b) = \{a \mid a \in P, (a, b, s, t) \in Q\}$$

Global trust

Ideally, when computing the trust in a certain peer, all the experiences with that peer should be taken into consideration. In this case a unique trust value will globally characterize each peer, independent of the observer. This is the global trust:

$$gt(b) = \alpha \sum_{(a,b,s,t) \in Q} s \cdot cr(a)$$

where α is a suitable normalizing factor and $cr(a)$ denotes the credibility of peer a who provided the feedback s. A similar global trust function is defined in [2], [50], and [46].

Starting from the assumption that peers are by default trustworthy, [3] records only negative experiences in form of complains. In this setting the global trust in a peer is computed based on the number of complains a peer has received. The lower this number the more trustworthy the peer is. The credibility of a peer as a source of feedback is also measured in terms of complains: the more complains have been filed by the peer the less trustworthy the source is. However, this interpretation might inhibit peers willingness to submit complains, as this is damaging their reputation.

Based on similar principles for global trust computation as [3], but departing from storing only complains, in [50] the credibility of a peer is computed based on its reputation. The higher the reputation the more credible is the peer for providing feedback.

On the other hand, in [46] the trust computation is very simple and does not take into consideration the credibility factor. However, in this system a peer reporting a feedback must provide in addition a proof of interactions. This limits the number of fake feedbacks as each feedback is at the cost of an interaction.

The global trust has the advantage of taking into account all the experiences that are available with a certain peer. However, in general, experiences are distributed to all the peers in the network and making them globally available requires special data management. For instance, [18] implements the data management by extending the Gnutella protocol with messages for handling queries and answers pertaining to trust. But this increases considerable the load of the network. To solve scalability issues, [3] employs a virtual data (tree) structure, called P-grid, to manage the reputational information. The P-grid efficiently locates and retrieves all the necessary information for computing the trust.

Local trust

Computing the global trust is not in general feasible [2]. Instead, each peer computes its own local trust in another peer. The local trust is computed as a function of the local experiences and the received recommendations.

The part of the trust which is dependent only on the own experiences is the direct trust:

$$dt(a,b) = \alpha \sum_{(a,b,s,t) \in Q_a} s$$

Taking in consideration only local available experiences is limiting. As shown in [49, 37, 30] exchanging recommendations helps peers learn more accurately the trustworthiness of the ones they intend to interact with. Therefore, direct trust is merged with the recommendations received from others. Ideally peer a, who computes the trust in peer b, should receive recommendations from all the peers who interacted with b. That is, peer a will get recommendations from all the peers in the set $W(b)$. However, in reality it is only a subset of the $W(b)$ that will eventually provide recommendations to a. We denote this subset as $W_a(b)$, where $W_a(b) \subseteq W(b)$. With this, the formulae for computing trust is:

$$t(a,b) = w_d * dt(a,b) + w_r * \beta \sum_{r \in W_a(b)} rt(a,r) \cdot rec(r,b)$$

where the w_d and w_r are the weights for the direct trust and the trust resulted from recommendations, and β is the normalization factor for the aggregation of recommendations. A trust computation function similar to this is proposed in [1, 3, 18, 49, 46, 50].

As pointed out in [50] it is reasonable to distinguish between trust in service providers and trust in recommenders, as in extreme case, a peer may maintain a good reputation by performing high quality services and sending malicious recommendations about her competitors. Thus, a ceratin recommendation should be combined with the trust that can be invested in the recommender before it is considered in the trust computation.

Other factors could influence the accuracy of the trust computation. For instance, [50] investigates the following factors:

- feedback in terms of amount of satisfaction,
- credibility of feedback,
- number of transactions,
- time when the reported transactions occurred,
- transaction context factor which allows for differentiating feedbacks from various types of transactions,
- community context factor as a peer may be considered to be more or less trustworthy depending on her social position.

The recommender trust can be computed based on the recommendations that the peer has previously given compared to the evaluated interaction. In [49] the recommender trust is computed as a learning function based on the previous trust and the outcome of the interaction. That is, $rt^{new}(a,r) = \gamma * rt^{old}(a,r) + (1-\gamma) * s$, where γ is the learning factor.

Although the computation of the recommendations is note explicitly stated by the existing systems, this is a sensitive aspect. Recommendations convey other peers' trust in a certain entity. Thus, the recommendation should be computed as trust is computed. However, it is important that recommendations coming from different peers are independent. They should not be based on the same set of experiences. Therefore, a requirement for the recommendation is that they should convey peers direct trust and they should be computed based only on the peer's own experiences. This means that $rec(a, b) = dt(a, b)$.

Aggregating complete opinions

When computing $t(a, b)$, the local trust computation will not take in consideration the opinions of all the peers that have interacted with peer b, that is the set $W(b)$, but only the opinions of a the "friends" of peer a (peers that a trusts as recommenders), that is the $W_a(b) \subseteq W(b)$.

However, it is possible to extend the trust computation from taking into account not only the friends of a but also the friends of the friends of a and so on [30]. All the peers can be seen as part of a trust graph, where the neighbors nodes of a peer a correspond to the peers that a trusts (see section 2.3.2).

The EigenRep reputation system, by Kamvar and colleagues, uses a distributed trust computation algorithm where the computation is iteratively carried by all the peers in the graph [30]. At each step in the algorithm, each node in the graph receives trust information from its predecessor nodes, merges this data with its own trust, and propagates the resulting data to its successor nodes in the graph. The computation has as many steps as the length of the trust chain. As shown in [30], if the global trust matrix is irreducible and aperiodic then the trust computed at each node will eventually converge towards a unique trust value for each peer in the system.

This approach has the advantage of merging the opinions of all peers in the system therefore giving a more accurate approximation of the trustworthiness of a certain peer. However, the approach introduces certain issues. The conditions under which the algorithms converges might be too strong for a real trust matrix. Also, the distributed algorithm requires tight synchronization of all peers, which might be a too strict requirement for the peer to peer network. Moreover, the trust computation convergence depends on the eagerness or laziness of nodes to propagate information.

Bayesian-network based trust computation

Trust depends on the context. That is, Alice trusts Bob as a car mechanic, but she might not trust him as an accountant. However, trust is also multi-faceted, even in the same context, and peers have to develop trust in different aspects of other peers [48]. For example, in a file sharing application one may be interested in different aspects characterizing the trustworthiness of a file providers, such as the file type, the file quality, or the download speed. Moreover, one aspect might be more important for one user and less important for another. Thus, every peer keeps a Bayesian network for each file provider she has interacted with.

For example, assume that $T = 1$ denotes a satisfying transaction and $FT = music$ the fact that the file involved contains music. Given the history of previous interactions, the following probabilities can be easily computed: probability $P(T = 1)$ of a transaction being satisfying; the probability $P(FT = music)$; and the conditional probability $P(FT = music \mid T = 1)$ denoting that the file type is *music* given the transaction is satisfying. Given these probabilities, one

can now compute, based on Bayes rules, the $P(T = 1 \mid \ FT = music)$ denoting the probability that a satisfying transaction will occur, provided that we are searching for music.

$$P(T = 1 \mid \ FT = music) = \frac{P(FT = music \mid \ T = 1) \cdot P(T = 1)}{P(FT = music)}$$

The Bayesian networks approach clusters peers with similar norms for trust evaluation. As shown in [48] this increases the rate of successful interactions. However, the requirement for each peer a to store one Bayesian network for each other peer b she interacts with raises serious scalability concerns.

2.3.2 Trust computation in the web of trust

The web of trust has been proposed primarily in the context of decentralized Public Key Infrastructures, such as the PGP. In such a system any peer can play the role of certificate authority and sign certificates binding the public keys to identities. The result is a web of trust where key holders can make publicly known whose keys they trust to be authentic and who they trust to be an introducer of new keys. However, not all peers can be trusted the same, and therefore trust levels can be assigned to links in the web of trust indicating the credibility invested in each link. Allowing the trust to be transitive and giving appropriate rules for aggregating trust values, one can compute the trust among any two nodes in the web of trust, even if not directly connected.

A similar scenario exists for the emerging Semantic Web. One major problem is that, by its nature, Semantic Web is a large and uncensored system to which any one can contribute. This raises the question of how much credence to give each source [43]. Keeping a central repository with trustworthiness of each participant is unfeasible. Instead, each peer in the Semantic Web develops and maintains trust relationships in a relatively small number of other peers she has encountered so far. All the peers form therefore a web of trust. In order for one peer to trust the statements made by another peer she does not directly trust, algorithms are proposed to traverse the web of trust and aggregate the trust values of intermediate links.

Trust transitivity

It has been argued that trust is not transitive [15, 1]. That is, if Alice trusts Bob, and Bob trusts Anna, it does not necessarily follow that Alice must trust Anna by any degree. This is because trust is within a certain context. If for example Alice trusts Bob to be a good doctor, and Bob recommends Anna to be a good doctor, Alice should not necessarily trust Anna to be a good doctor unless Alice trusts Bob to be a good recommenders of doctors. But "trust as good doctor" and "trust as a good recommender for a doctor" have different contexts.

This is further complicated if Alice trusts Bob to be a good recommender for a doctor, and Bob recommends Anna to be a good recommender for a doctor. Should Alice trust the recommendations from Anna? According to the argument above she should not. However, a more compact and general interpretation of the trust as recommender can be adopted [29]: if Alice trusts Bob as a recommender she will also trust Bob to recommend a recommender for a doctor. In this case recommender trust becomes transitive.

Some authors, e.g. [14, 43], do not distinguish the trust for a context from trust for recommending for a context. Instead they consider that trust is, in general, transitive. That is, if Alice trust Bob, and Bob trusts Anna, then Alice trusts Anna. As shown in [14], the distinction

between the two different types of trust do not change the complexity of the trust computation problem.

In general trust transitivity enriches the possibilities to get more recommendations. If Alice trusts Bob as a recommender, and Bob recommends Anna as a recommender, Alice, by trusting Anna as recommender, has the opportunity to reach the experiences of all the entities that Anna trusts. These experiences would have been remained unknown or untrussed to Alice if the recommender trust was not transitive. Trust transitivity allows one peer to "walk the web of trust" [14] and find new recommendations that enriches the knowledge about other peers, allowing for a more accurate trust computation.

Computational models in the web of trust

The main issues to be addressed are: what is the semantic of trust and how to compute trust in the web of trust?

In general, a probabilistic semantic of trust has been preferred in the context of web of trust. For instance in [14] the trust value assigned to a link in the web of trust is a probability indicating the degree of belief a peer has in the fact that a certain public key is indeed bound to a certain peer. Similarly, in [43] links in the web of trust are probabilities indicating either peers' personal beliefs in semantic statements or peers' personal trusts in other peers. Probabilistic interpretations of the trust value are given also in [7, 29].

On the other hand, for the trust computation both probabilistic and non-probabilistic interpretations have been given.

Probabilistic interpretation

The trust computation has been split into trust computation along a trusting path (serial trust path) and trust computation across multiple parallel paths (parallel trust path). For serial and parallel trust paths the probabilistic computation is intuitive and efficient. However, for a more general topology of the web of trust, including interconnected paths and cycles, the probabilistic interpretation exists but the computation is very expensive. We show the probabilistic interpretation, as proposed in [14], by first introducing some simple examples for the serial, parallel, and interconnected trust paths.

Serial trust path

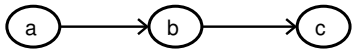


Figure 2.8: Serial trust paths

Figure 2.8 shows a very simple serial path topology for the web of trust. In terms of probabilities the two trust links corresponding to two independent events (ab and bc), where the final trust a has in b corresponds to the intersection of these two independent events ($ab \cap bc$). Following the rules of probability, the trust a has in c is the result of the product of the serial trusts, that is:

$$t(a, c) = t(a, b) * t(b, c)$$

.

Note that the trust in the target decreases along the serial path when more entities lie between the start and the target node.

Parallel trust path

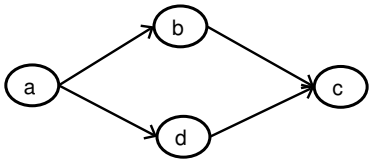


Figure 2.9: Parallel trust paths

A parallel trust path topology is depicted in figure 2.9. In this case the trust of a in c corresponds to the event $(ab \cap bc) \cup (ad \cap dc)$. Note that this is the union of two independent paths and thus following the rules of probability the final trust is:

$$t(a,c) = 1 - (1 - t(a,b) * t(b,c)) * (1 - t(a,d) * t(d,c))$$

Interconnected paths

The serial and the parallel topologies are idealizations. In reality the trust topology may contain interconnected paths and cycles which complicates the trust computation. For instance, figure 2.10 shows the previous parallel paths scenario but now with the two paths interconnected by the link from b to d.

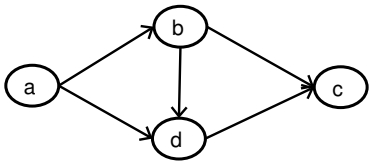


Figure 2.10: Interconnected trust paths

In this case the resulting trust a has in c corresponds to the event $(ab \cap bc) \cup (ab \cap bd \cap dc) \cup (ad \cap dc)$. This event is the concatenation of all the possible paths from a to c. However, the paths are not independent of each other and therefore the trust can not be computed iteratively as it was the case for the serial and parallel trust scenarios.

Although some of the proposed probabilistic computation models, e.g. [27], fail to deal with interconnected paths, a pure probabilistic interpretation of the general trust topology is possible. A generalized probabilistic algorithm for computing trust is given in [14]. The algorithm is capable to deal with any kind of trust graph. However, as shown there, the theoretical algorithm for computing trust based on the probabilistic interpretation is of exponential complexity for the general case. Similar results have been shown in [7] where the probability interpretation was only partially followed.

Heuristics for trust computation

The probabilistic interpretation of trust computation leads to exponentially complex algorithms. As an alternative, many non-probabilistic interpretations of trust computation have been proposed [14]:

- The worst path. Only consider the worst available path leading from the start node to the target. This is a pessimistic approach which discredits all the possible good paths.
- The best path. Only consider the best available path leading from the start node to the end node. This is probably an overoptimistic approach. Note that both the worst and the best path are poorly resilient to malicious recommenders. A malicious recommender can control the computation path by giving very bad or very good recommendations.
- Find all independent paths and calculate their mean value. Although simple, intuitive, and efficient to compute, the mean breaks the probability interpretation.
- Hull. Select the best path from start to the target, calculate and remove it. Repeat until no more paths can be found. This assures that all the independent paths have been found. Now combine these paths by distrust. The approach keeps up with the probability interpretation by ignoring some of the links in the graph.
- All. Calculate the probability of all possible paths. Combine them via distrust. Unfortunately, this breaks the independence assumptions from the probability laws. Considering all path between the start and the target nodes resembles the path algebra interpretation from [43] and the trust computation proposed in [51]. However the latter two approaches use different heuristic for aggregating the paths in the final trust value, which depart from the probability interpretation. [43] uses the sum of all paths while [51] uses a variation of the simple maximum function.
- Interleaving. This is an iterative process which starts from the start node, computes trust in the neighbors of the start node, then computes trust in the neighbors of the neighbors and so far until the computation reaches the target node. If the trust graph suffice ceratin laws of independence then interleaving is an acceptable heuristic. The approach is fast and of practical use [14]. However, it is not exact, but it tends to be overly optimistic when trust paths contain looping structures or are otherwise interconnected. This heuristic is similar to the random walk interpretation from [43].

Path algebra interpretation

Borrowing from the generalized transitive closure algorithms, Richardson and colleagues [43] proposed a path algebra interpretation of the trust computation in the web of trust. Idea is similar to the "All" approach introduced by Caronni [14] (and described above) and it is based on finding all the paths from the start to the target node. However, [43] generalizes the operators for serial and parallel trust computation as follows: $\circ$ is the concatenation of trusts along a path, and $\diamond$ is the aggregation of trusts from different paths (resulted by applying the $\circ$ operator). Some possible concatenation functions are the multiplication, minimum, or maximum of the links along a trust path. Some possible aggregation functions are the addition, maximum, or minimum. The selection of the concatenation and aggregation operators may

depend on the application domain, desired trust, the cyclic semantics of the web of trust, and the expected social behavior in the domain.

Note that the algorithm enumerating all the possible paths from a start to a target node is feasible only if the topology of the web of trust is globally known. However, this might be unrealistic. Therefore, [43] propose a distributed algorithm which iteratively computes the trust in the neighbors, and then the trust in the neighbors' neighbors and so far. This approach is similar to the "Interleaving" algorithm proposed by Caronni. Defining $\bullet$ as being the combination function incorporating both the concatenation and the aggregation, the trust can be computed iteratively as:

$$\mathcal{T}^{(0)} = \mathrm{T}, \mathcal{T}^{(n)} = T \bullet \mathcal{T}^{(n-1)}, repeat\ until\ \mathcal{T}^{(n)} = \mathcal{T}^{(n-1)}$$

where $\mathcal{T}^{(i)}$ is the computed trust at the i^{th} iteration, while T is the local trust vector.

The combination of trust will converge on acyclic graphs. For cyclic graphs the combination function will converge only if the combination is cyclic-indifferent. That is, the combination is not affected by the introduction of a cycle in the path between two peers. As shown by the authors, the multiplication and the maximum are cyclic-indifferent aggregation functions, whereas minimum and average are not.

Random walks interpretation

The path algebra interpretation requires the combination function to be cyclic-indifferent for the algorithm to converge. However, this might be a too restrictive requirement. Instead, a probabilistic interpretation based on the random walks on Markov chains has been proposed in [40] and similarly in [43]. A surfer x searching for trustworthy peers can crawl the network using the following rule: at each peer a, it will crawl to peer b with the probability $t(a, b)$. After crawling for a while in this manner, the surfer is more likely to be at a reputable peer then at an unreputable peer. Departing from the random surfer algorithm, [43] also considers the so called *self trust* of the surfer - that is at each peer a the surfer will jump back to the initial node x with a probability equal to $t(x, x)$.

In this interpretation the combination function can be non cyclic-indifferent and the algorithm will still converge. The random walk approach, used in a simplified version in PageRank, has been found to be of practical use in discovering high-quality web pages [40].

Nonmonotonicity of the trust computation

The aggregation of trust values has the property of being nonmonotonic. The computed trust can increase or decrease as new trust relationships are added.

In general, it is unknown whether all possible trust relationships have been derived and given as input to the computation. Thus, an estimation of the possible effects of new relationships on the combined trust value is necessary (by taking into account the probability density). Otherwise, when using the "incomplete" combined trust value, one can but hope that no significant changes will appear [7].

Trust relationship specification

Deriving new trust relationships starting from the exiting ones should be done based on certain rules. However, most of the proposed models use implicit, or informally defined, derivation

rules. Notably, the early work by [7] has explicitly and formally defined rules for deriving trust relationships. Moreover, the model allows for specifying constants on the trust relationships.

The model allows for specifying direct trust (that is trust in a context) and recommendation trust relationships as follows.

Direct trust

a $trusts_x^{seq}$ *b value v*

A direct trust relationship exists if all experiences with b with regard to trust class x which a knows about are positive experiences. *seq* is the sequence of entities that mediated the experiences (i.e. the recommendation path)excluding a and b. v is the value if the trust relationship which is an estimation of the probability that b behaves well when being trusted.

Recommendation trust

a $trusts.rec_x^{seq}$ *b when.path* S_p *when.target* S_t *value v*

A recommendation trust relationship exists if a is wiling to accept reports from b about experiences with third parties with respect to trust class x. This trust is restricted to experiences with entities in S_t (the target constraint set) mediated by entities in S_p (the path constraint set). v is the value of the trust relationship. It represents the portion of offered experiences that a is willing to accept from b and is based on the experiences a has had with the entities recommended by b.

Beth and colleagues [7] also make explicit the rules for deriving trust and recommender trust.

Rule 1 (new direct trust):

$a\ trusts.rec_x^{seq_1}\ b\ when.path\ S_p\ when.target\ S_t\ value\ v_1$
$\wedge\ b\ trusts_x^{seq_2}\ c\ value\ v_2$
$\wedge\ c \in_s S_t$
$\wedge \forall X : (X \in_l seq_2 \Rightarrow (X \in_s S_p \wedge X \notin_l a \circ seq_1))$
$\Rightarrow atrusts_x^{seq_1 \circ b \circ seq_2}\ c\ value\ (v_1 \odot v_2)$

Rule 2 (new recommendation trust):

$a\ trusts.rec_x^{seq_1}\ b\ when.path\ S_{p1}\ when.target\ S_{t1}\ value\ v_1$
$b\ trusts.rec_x^{seq_2}\ c\ when.path\ S_{p2}\ when.target\ S_{t2}\ value\ v_1$
$\wedge \forall X : (X \in_l seq_2 \circ c \Rightarrow (X \in_s S_{p1} \wedge X \notin_l a \circ seq_1))$
$\Rightarrow atrusts.rec_x^{seq_1 \circ b \circ seq_2}\ c$
$when.path(S_{p1} \cap S_{p2}) when.target(S_{t1} \cap S_{t2})\ value\ (v_1 \cdot v_2)$

The symbol $\circ$ denotes concatenation of sequences, $\in_l$ denotes membership to a sequence, and $\in_s$ denotes the membership to a set. The operator $\odot$ is defined as follows: $v_1 \odot v_2 = 1-(1-v_2)^{v_1}$.

The authors show that the choice of the derivation strategy has no influence on the resulted trust expressions. That is, from a given sequence of trust expressions one can either derive a unique trust expression or none at all, independent of the order in which the rules are applied.

To track all the peers which can be trusted by a peer p with respect to a trust class, one has to go along all noncyclic paths in the network which consists of trust relationships of the desired class, start at a, follow the collected constraints and end with a direct trust relationship. A trust derivation algorithm is proposed which has exponential complexity in the number of nodes and the problem has been proven to be NP-complete. To remedy this, heuristics have been proposed. A distributed algorithm is proposed which can handle all types of networks

but it is specially designed for tree-like structures. For pure tree structures the algorithm's complexity is logarithmic.

Logic based calculus

Josang [27] proposes an algebra for determining trust in certification chains. The algebra is based on Subjective Logic and it can cope with uncertainty. The assumption is that one can not always have a belief or a disbelief with regard to a certain statement. Instead, one can be in the position of not knowing. Therefore in [27] each statement is assigned an opinion which is defined as the triplet consisting of: b (a measure of one's belief), d (a measure of one's disbelief), and u (a measure of uncertainty), such that $b + d + u = 1$, and $b, d, u \in [0, 1]$. Subjective logic defines various logical operators for combining trust. The operators for aggregating trust along the paths and across paths are defined in terms of probability, similar to what we have discussed in section 2.3.2. However, the system is not capable to completely analyze and compute trust in the general case where interconnected paths exist.

Josang and Grandison [28] augments the Subjective Logic with a new operator, the *conditional inference*, highlighting the usefulness of subjective logic over binary logic and probability calculus because it can model situations where the antecedent, the consequent and the conditional itself are uncertain. That is, given that statement x has the associated opinions o_x and the conditional $x \rightarrow y$ (if x then y) has the opinion $o_{x \rightarrow y}$, the problem is to find o_y, the opinion of the conclusion y.

2.3.3 Open problems and future work

Trust computation models have the advantage of being highly sensitive to the dynamics of peer's behavior. However, for the computed trust to accurately reflect the trustworthiness of peers it needs to take in consideration all the relevant feedback existing in the system. Therefore, when using the incomplete knowledge, one should also consider the probability density and estimate the possible changes induced by the data that is missing.

Trust computation for the web of trust can gather opinions of the friends of the friends and so on, thus increasing its accuracy. Pure probabilistic interpretation of the computation is possible, but the algorithm is of exponential complexity. Instead, a number of heuristics have been investigated, from which the path algebra and the random walk interpretations are the most promising. However, these heuristics raise the questions of convergence and synchronization. For the algorithms to converge it is necessarily that the trust graph is irreducible and aperiodic, properties that are not necessary satisfied by a real trust graph. Also, the convergence of the algorithm depends very much on the synchronization of the peers and on their willingness to actually perform the computation and propagate the results. Therefore, a reputation-based trust solution would need to include incentives for peers to collaborate and be truthful.

In order for the reputation systems to be used in large open distributed systems well defined semantics for trust information as well as for the trust computation should be established. Also, the trust computation should be integrated better with other security decision function, such as access control. In addition, languages for querying, retrieving, and inserting reputational information as well as scalable and efficient management of this data needs to be provided by the underlying data management infrastructure.

Bibliography

[1] A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *HICSS 2000*, 2000.

[2] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *Ninth International Conference on Cooperative Information Systems (CooIS 01)*, 2001.

[3] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *CIKM 2001*, pages 310–317, 2001.

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

[5] 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.

[6] 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.

[7] T. Beth, M. Borcherding, and B. Klein. Valuation of trust in open networks. In *Proceedings of the Third European Symposium on Research in Computer Security*, pages 3–18. Springer-Verlag, 1994.

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

[9] 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.

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

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

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

[13] P. Bonatti and P. Samarati. Regulating Service Access and Information Release on the Web. In *Conference on Computer and Communications Security*, Athens, Nov. 2000.

[14] G. Caronni. Walking the web of trust. In *WETICE*, 2000.

[15] Bruce Christianson and William S. Harbison. Why Isn't Trust Transitive? In T. Mark and A. Lomas, editor, *Security Protocols, International Workshop 1996*, LNCS 1189, pages 171–176, Cambridge, United Kingdom, April 10-12 1996. Springer.

[16] 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.

[17] 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.

[18] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *ACM Conference on Computer and Communications Security*, pages 202–216, 2002.

[19] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st First European Semantic Web Symposium*, Heraklion, Greece, May 2004.

[20] J. Golbeck, B. Parsia, and J. Hendler. Trust networks on the semantic web. In *Cooperative Intelligent Agents*, Helsinki, Finland, August 2003.

[21] T. Grandison. *Trust Management for Internet Applications*. PhD thesis, Imperial College London, 2003.

[22] 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.

[23] 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.

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

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

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

[27] A. Josang. An algebra for assesing trust in certifiction chains. In *Network and Distributed Systems Security Symposium 1999*, 1999.

[28] A. Josang and T. Grandison. Conditional inference in subjective logic. In *6th International Conference on Information Fusion*, 2003.

[29] A. Josang, E. Gray, and M. Kinateder. Analysing topologies of transitive trust. In *Workshop of Formal Aspects of Security and Trust*, 2003.

[30] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. Eigenrep: Reputation management in p2p networks. pages 640–651, 2003.

[31] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[32] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust-management Languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 58–73, January 2003.

[33] 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.

[34] N. Li, J. Mitchell, and W. Winsborough. Design of a Role-based Trust-management Framework. In *IEEE Symposium on Security and Privacy*, Berkeley, California, May 2002.

[35] N. Li, W. Winsborough, and J. Mitchell. Distributed Credential Chain Discovery in Trust Management. *Journal of Computer Security*, 11(1), Feb. 2003.

[36] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition edition, 1987.

[37] S. Marti and H. Garcia-Molina. Identity crisis: Anonymity vs. reputation in P2P systems. In *Peer-to-Peer Computing 2003*, pages 134–141, 2003.

[38] W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: automated trust negotiation for peers on the semantic web. In *Workshop on Secure Data Management in a Connected World (SDM'04)*, Toronto, Aug. 2004.

[39] 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*, San Diego, 2004.

[40] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[41] A. A. Pirzada and C. McDonald. Establishing trust in pure ad-hoc networks. In *Proceedings of the 27th conference on Australasian computer science*, pages 47–54. Australian Computer Society, Inc., 2004.

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

[43] M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In *Second International Semantic Web Conference*, September 2003.

[44] D. Romano. *The nature of trust: clarification of its defining characteristics*. PhD thesis, Louisiana State University, 2002.

[45] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobsen, H. Mills, and L. Yu. Requirements for Policy Languages for Trust Negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks*, Monterey, CA, June 2002.

[46] A. Singh and L. Liu. TrustMe: Anonymous Management of Trust Relationships in Decentralized P2P Systems. In *Peer-to-Peer Computing 2003*, pages 142–149, 2003.

[47] J. Trevor and D. Suciu. Dynamically distributed query evaluation. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Santa Barbara, CA, USA, May 2001.

[48] Y. Wang and J. Vassileva. Bayesian network trust model in peer-to-peer networks. In *Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003)*, 2003.

[49] Y. Wang and J. Vassileva. Trust and reputation model in peer-to-peer networks. In *Peer-to-Peer Computing 2003*, pages 150–157, 2003.

[50] L. Xiong and L. Liu. PeerTrust: Supporting Reputation-Based Trust in Peer-to-Peer Communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, July 2004.

[51] B. Yu and M. P. Singh. A social mechanism of reputation management in electronic communities. In *Proceedings of the 4th International Workshop on Cooperative Information Agents IV, The Future of Information Agents in Cyberspace*. Springer-Verlag, 2000.

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

Chapter 3

Action Languages

3.1 Introduction

Reasoning about action and change is a kind of temporal reasoning where, instead of reasoning about *time* itself, we reason on *fenomena* that take place in time.

Indeed, theories of reasoning about action and change describe a *dynamic world* changing because of 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 true 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 problem in knowledge representation theory.

Different approaches in literature took different assumptions on temporal ontology and then they developed different abstraction tools to cope with dynamic worlds. However, most of formal theories for reasoning about action and change (*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*. These assumptions are listed below:

- the dynamic world that the theory aims to model 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's 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 execution of actions. To be precise, in general, 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 precondition 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 in later states, i.e after some action's execution.

In the following, the term *domain descriptions* 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 actual state (reasoning from causes to effect). 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 may 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 might contain information on (a), (b) and (c), then the logical framework might 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).

An important 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 hold in the state resulting from the a's execution.

A second formalization difficulty, known as the *ramification problem*, arises in the presence of the 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 [47, 42] and those addressing the problem by using non-classical logics [56, 16, 62, 29] or computational logics [27, 12, 45, 6]. In the following, we will briefly review the most popular logic-based approaches to reason about action and change.

3.2 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 [47] to capture change in first order classical logic. 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. Later on, Kowalski and Sergot have developed a different calculus to describe change [42], 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. Modal logics adopts 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.

Both situation calculus and modal logics influenced the design of logic-based languages for agent programming. In the following we will describe a high-level robot programming language, called GOLOG, based on a theory of actions in the situation calculus, and the logic programming language DyLOG based on modal logic

3.2.1 Situation Calculus

The situation calculus [47] was designed for representing dynamically changing worlds in first order classical logic. All changes to the world are the result of the execution of *actions*. 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 actions. Preconditions can be represented with a predicate $Poss$, as in:

$$Poss(pickup(x, s) \equiv [\forall z \neg holding(z, s)] \wedge nexto(x, s) \wedge \neg heavy(x)$$

whereas effects can be specified as:

$$Poss(drop(r, x), s) \wedge fragile(x, s) \Rightarrow broken(x, do(drop(r, x), s))$$

meaning that dropping a fragile object causes it to be broken, or

$$Poss(repair(r, x), s) \Rightarrow \neg broken(x, do(repair(r, x), s))$$

meaning that repairing an object causes it to be not broken.

3.2.2 Modal approaches

The suitability of dynamic logics or modal logics to formalize reasoning about actions and change has been pointed out in various proposals [19, 56, 16, 62, 29]. Modal logics adopts 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 make use of 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:

$\Box(loaded \Rightarrow [shoot]\neg alive)$

$\Box[shoot]\neg loaded$

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.

Preconditions might be represented as:

$\Box(\neg have\_gun \Rightarrow [shoot]\bot)$

meaning that action *shoot* cannot be executed if the fluent *have_gun* is false.

One of the most challenging problems in reasoning about actions is *ramification*, i.e. the problem of dealing with indirect effects of actions. It rests on the concept of causality, which has been widely studied in philosophy, and for which several theories have been proposed. For instance we want to be able to express rules where $\neg alive$ causes $\neg walking$ for the previous example. Thus, if the gun is initially loaded, and Fred is alive and walking, after *shoot*, $\neg alive$ will hold, and $\neg walking$ as well, according to the above causal rule.

In general we do not want to represent causal rules with material implication because it has undesired properties such as contraposition. In our example we do not want to use the causal rule to derive *alive* from *walking*, because we do not expect the truth value of the fluent *alive* to be influenced by the one of *walking*. Thus any approach to causality should avoid using contrapositives of causal rules. A solution proposed in [30] is to introduce a new modal operator ©, which blocks contraposition of implication, such that the above causal rule will be represented as:

$\Box(\neg alive \Rightarrow ©\neg walking)$.

An action theory expressed in modal logic can be enriched by making use of *dynamic* or *temporal logics*. Dynamic logic allows to reason about complex actions, by expressing them as regular expressions on the alphabet of primitive actions. A related approach, used by language DyLOG, will be presented below. Temporal logic allows to formulate time related properties and constraints. For instance, the need of temporally extended goals has been motivated by Bacchus and Kabanza [1] and by Kabanza et al. [40], who proposed an approach to planning based on a linear temporal logic. This approach allows to formulate general goals, such as achievement and maintenance goals.

3.2.3 The frame problem

As pointed out by McCarthy and Hayes [47] 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. Various approaches have been proposed. 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 [58] proposed a solution which rests 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. For instance, the *successor state axiom* for fluent *broken* will be:

$Poss(a, s) \Rightarrow [broken(x, do(a, s)) \equiv$
$(\exists r(a = drop(r, x)) \wedge fragile(x, s)) \vee$
$broken(x, s) \wedge \neg\exists r(a = repair(r, x))]$.

[30] adopt a nonmonotonic approach, by adding *persistency assumptions* of the form: $[a_1; a_2; \ldots; a_n](l \Rightarrow [a]l)$. The basic idea is that persistency of a fluent from a state to the next one must be assumed, if it does not lead to an inconsistent state. Therefore the above persistency assumption must be assumed unless we are able to prove that $\neg l$ holds after execution of action a, i.e. unless we are able to prove $[a_1; a_2; \ldots; a_n; a]\neg l$. This solution was proved equivalent to using default logic.

3.3 Computational Logic

Non-classical logics have been successfully 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 mainly 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 among the researchers is growing the interest on the use of *computational logic*, which allow one to express formal specifications that can be directly executed, thanks to the fact that logic programs have a procedural interpretation, beside the declarative one [67].

3.3.1 The $\mathcal{A}$ family

In '93 Gelfond and Lifschitz have defined a simple declarative language for describing actions, called $\mathcal{A}$ [27]. Various extensions of $\mathcal{A}$ have been proposed in the last years with the intention to deal with nondeterministic actions [41, 9], concurrent actions [10], ramifications [41, 34] or sensing actions [11, 45, 12]. Most of the times, a sound translation of such extensions into logical languages is provided[1].

In most of these formulations, the target is to define a logical entailment relation between a domain description $\mathcal{D}$ (that contains causal laws for the domain's actions and observations on fluents value in the initial state) and simple *queries* of the form "f **after** $a_1, \ldots, a_n$" where f

[1]For the language $\mathcal{AR}_0$ defined in [41] a translation is given into a formalism based on circumscription, rather than into logic programming.

is a fluent and $a_1, \ldots, a_n$ are elementary actions:

$$\mathcal{D} \models f \ \textbf{after} \ a_1, ..., a_n$$

For instance, the following domain description for the shooting problem:

initially $\neg loaded$

initially $alive$

$Load$ **causes** $loaded$

$Shoot$ **causes** $\neg alive$ **if** $loaded$

$Load$ **causes** $\neg loaded$

entails

$\neg alive$ **after** $Load; Wait; Shoot$

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 negation as failure of logic programming.

Also more general queries have been considered where beside to reason about the effects of sequences of simple actions, it is possible to reason about conditional or complex plans execution. It is the case in the works [11, 45, 12] where the problem of extending the Gelfond and Lifschitz' language $\mathcal{A}$ for reasoning about complex plans in presence of sensing and incomplete information has been tackled.

3.4 Dealing with Incomplete Knowledge

In a pioneering work of '85 [51], Robert C. Moore was one of the firsts to recognize the central role the agent's knowledge plays in acting and achieving goals, especially considering that in the real world planning and acting must be performed without complete knowledge about the situation. "When the agent entertains a plan for achieving some goal he must consider not only whether the physical prerequisite for the plan have been satisfied, but also whether he has all the *information* necessary to carry out the plan" [51]. If it has not all this knowledge, the agent may need to have at its disposal *knowledge-producing actions* (also called *sensing actions*),that allow to acquire new information and, then, affect the mental state of the agent (instead of affecting the world state).

Moore proposed a formal theory of action and knowledge based on first-order logics. In his theory, in order to deal with incomplete information, he introduced a distinction between the state of the world and the state of the agent's knowledge. Furthermore, beside uninformative actions, he allows of sensing actions to acquire new information. To the best of our knowledge, it was the first theory allowing to represent and reason about *mental effects* of actions, beside of *world effects*. Moreover, note that Moore's model copes not only with mental effects of sensing-actions but mental effect of non-sensing actions as well. Indeed, he pointed out that even if an action is not informative, i.e. it does not provide an agent with new information, performing the action will still alter the agent's epistemic state. In fact, since the agent is aware of its action, it will know that it has been performed. "As a result, the tense and modality of many of the things he knows will change": if, for instance, before performing the action he knows

that a fact f will hold after performing the action, then after the execution of the action he will know that f is true.

Such concepts were represented by Moore in terms of possible world using first-order logics. Later, Scherl and Levesque [61] adapted the possible world model of knowledge proposed by Moore to the situation calculus.

In [22] De Giacomo and Rossati present a minimal knowledge approach to reasoning about actions and sensing in presence of incomplete information. Their proposed formalism combines the modal μ-calculus and autoepistemic logic.

In a recent work, Thielscher [66] faces the problem of representing a robot's knowledge about its environment in the context of the Fluent Calculus, a formalism for reasoning about actions based on predicate logic. In order to account for knowledge, basic fluent calculus is extended by introducing the concept of possible world state and defining the knowledge of a robot in terms of possible states. The formalism deals with sensing actions and it allows to distinguish between state of the world and state of knowledge of an agent about the world.

In [45] Lobo et al. introduce the language $\mathcal{A}_K$ which provides both actions to increase agent knowledge and actions to lose agent knowledge. It has a general semantics in which epistemic states are represented by sets of worlds. Complex plans are defined as Algol-like programs containing sequences, conditional statements and iteration. Given a domain description in $\mathcal{A}_K$, a query of the form ϕ **after** $[\alpha]$ is true if ϕ holds in every model of $\mathcal{D}$ after the execution of the plan α in the initial state, where α is a complex plan, possibly including conditionals and iterations.

In [12] Baral and Son define an action description language, also called $\mathcal{A}_K$, which deals with sensing actions and distinguishes between the state of the world and the state of knowledge of an agent about the world. The semantics of the language are proved to be equivalent to the one in [45] when rational models are considered.

The action language presented in [8] defines a language capable of representing incomplete belief states and of dealing with sensing actions. An epistemic level is introduced in the action logic in order to represent the mental state of an agent, by using belief modalities. As concerns world actions, i.e. actions affecting the real world, the language model what the agent believes about actions effects based on its beliefs about the preconditions. Sensing actions are considered as input actions which produce fresh information on the value of some fluents in the real world. In essence, sensing actions are regarded as non-deterministic actions, whose outcome cannot be predicted by the agent.

3.5 Logic-based agent languages

The theory of computational agents plays a central role in AI, providing powerful conceptual tools for characterizing complex software systems situated in dynamic environments where they possibly interact with other computational entities. In the literature there is a wide agreement in defining agents as intelligent systems toward which we take the *intentional stance* [24]. This is done by attributing agents with cognitive concepts such as beliefs and goals in order to describe, analyze or predict their behaviour. Moreover, *software agents* are usually designed as computational entities exhibiting

- high-degree of *autonomy*,
- capability of pursuing their goals eventually by interacting with other software or humans (*proactiveness*), and

- capability of getting feedback on changes which occur in the environment they are situated in (*reactivity*).

One of the core research issues in the community dealing with agents is the design of *knowledge representation languages* for specifying and reasoning about the internal behavior of an agent, as well as about the dynamic change of its mental state.

In general, modeling agent's internal behaviour and attitude dynamics is a difficult task. In particular, many theories of agency have been proposed which are based on *logic formalisms*. In different ways, they all try to define a formal model including accounts of the individual agent's general reasoning and behaviour strategies.

One of the technical difficulties regards the ability to manage incomplete and multiple knowledge. Indeed, generally 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 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 (as 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.

Nonetheless, due to the technical difficulty of verifying that those properties granted by the formal models are granted also in the practical systems that implement them, there is a gap between expressive power of formal models of agency and practical implementations. One way of filling the gap between agent theories and agent system's implementation is to use *computational logic* which supports *logic-based executable agent specifications* and facilitates 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.

Starting from such premises, computational counterparts of non-classical logics seem to be a promising candidate as agent specification languages [8, 59]. Indeed, in non-classical logics it is easier and more natural to describe systems which involve notions such as knowledge, beliefs and reasoning about actions. Moreover, the logic programming framework offers efficient execution mechanisms for the language retaining its desirable properties such as its declarative semantics and high-level descriptive capabilities.

Both situation calculus and modal action logics influenced the design of logic-based languages for agent programming. On the one hand, recently the research about situation calculus gained a renewed attention thanks to the cognitive robotic project at University of Toronto. This project has lead to the development of a high-level agent programming language, called GOLOG, based on a theory of actions in situation calculus [44]. On the other hand, in [8], a modal action theory has been used as a basis for specifying and executing agent behaviour in a logic programming setting. Finally, the language IMPACT is an example of use of deontic logic for specifying agents. The agent's behavior is specified by means of a set of rules (the agent program). These rules are suitable to specify, by means of deontic modalities, agent policies, 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.

A review of such languages is given in section 3.6.

3.5.1 Linear and conditional plans

Agent settled in dynamic environments must be able to perform practical reasoning, as deciding what state of affairs it wants to achieve (deliberation) and deciding how it is going to achieve this state of affairs (means-ends reasoning). Means-ends reasoning is the process of deciding how to achieve an end (i.e. a goal) using the available means (i.e. the actions which can be performed). This process is also known as *planning*. Typically an agent uses some function plan(B,G) which determines a plan π to achieve the goal G.

An agent could engage in the generation of a plan from scratch, as in standard planning in AI. For efficiency reasons, in most agent systems, the plan function is implemented by giving the agent a *library of plans*, which have been predefined by the designer of the agent. 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 will be sound given the the agent's current beliefs.

In some approaches [8, 44] the starting point is not a simple set of atomic action definitions, but a procedure defining a complex action the agent can perform for achieving its tasks. In this case the planning task is interpreted as finding a terminating execution of the procedure that, when executed, leads to a state where the desired goal holds. Such terminating execution will be a legal *sequence of actions* and a *correct plan* w.r.t the goal. It easy to see that this is a special case of the planning problem, where the procedure definition constrains the search space in which to look for the wanted sequence

The issue of formulating the planning task in dynamic domains including *sensing* was formulated by Levesque in [43]: "A number of researchers are investigating the topic of conditional planning. [...] where the output for one reason or another is not expected to be a fixed sequence of actions, but a more general specification involving conditionals and iteration. In this paper we will concern with conditional planning problems where what action to perform next in a plan may depend on the result of an earlier *sensing actions*."

The Guardian of the Room Example *A room has only two sliding doors, door*1 *and door*2*, leading outside. In the initial situation the robot is inside the room close to door*2*. It is possible to go from a door to another. Being close to a door, it is possible to close or open it, by toggling the switch next to the door: if the door is open, by toggling the switch the robot will close it and viceversa. Moreover, robot can check if a door is open by activating its sensors. The goal is to close all doors.*

A linear plan (i.e. a sequence of actions) cannot solve this problem. Indeed both in case it has incomplete knowledge on the door state, and in the case it cannot exclude that someone has closed it since the last time it checked (i.e. an exogenous action has occurred), the robot could not know in advance whether the door is open or not. Then, it has to be able to condition its course of actions on the runtime result of sensing. What we expect in this setting is a complex (conditional) plan that leads to a goal state no matter the sensing turns out. For our example an expected solution could be something like:

```
/* Assuming the robot close to door2 initially */

check the state of door2;
if door2 is open
        then close it;
             go to door1;
             check the state of door1;
             if door1 is open
                  then close it;
                  else do nothing;
else go to door1;
```

check the state of $door1$;
if $door1$ is open
then close;
else do nothing.

3.6 Executable agent specification languages: literature

Logic-based executable agent specification languages have been deeply investigated in the last years. In this section we will briefly recall the main features of three of them - GOLOG [44], DyLOG [8] and IMPACT [59]-, which seems to be particularly promising as implementation languages for intelligent applications in the semantic web context.

3.6.1 GOLOG: ALGOL in logic

GOLOG is a programming language, developed at the University of Toronto, for the specification and the execution of complex actions in dynamic domains. It is a procedural language mainly designed for applications as programming high-level robot control and intelligent software agents. Recently it has been used as high-level formalism for automatically composing services on the semantic web in the context of the DAML-S initiative [49, 50]. GOLOG is based on a logic for actions expressed in an extended version of the situation calculus. The meaning of primitive actions is specified in situation calculus by giving their preconditions and effects in terms of suitable axioms (see section 3.2.1), while larger programs are defined by macro specifications which expands into (sometimes second order) formulae of the situation calculus and allow to assemble primitive actions into complex actions. In particular, complex actions are defined using the abbreviation $Do(\delta, s, s')$ where δ is a complex action expression; intuitively $Do(\delta, s, s')$ will hold whenever the situation s' is a terminating situation of an execution of δ starting in situation s. Constructs for building complex actions include the following:

a - primitive action
$\delta; \delta$ - sequence
$\phi?$ - test
$\delta_1 | \delta_2$ - non deterministic choice between actions
$(\pi x)\delta(x)$ - non deterministic choice of arguments
δ^* - nondeterministic iteration

Formalization of complex actions draws considerably from dynamic logic. It reifies as situations in the object language of the situation calculus the possible worlds with which the semantics of dynamic logic is defined (see [44] for details). GOLOG attempts to blend ALGOL programming style into logic. In fact, on the one hand, it borrows from ALGOL very well-studied programming constructs as sequence, conditionals, recursive procedure and loops. For instance:
if ϕ **then** δ_1 **else** $\delta_2 =_{def} [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$
It is also possible to define recursive procedures, whose semantics is given as least fixed-point. The following is a generic procedure for making travel arrangement:

proc $Travel(cust, origin, dest, dDate, rDate)$;
if $registrationRequired$
then $Register$ **endIf**;
$BookTransport(cust, origin, dest, dDate, rDate)$;

$BookAccomodations(cust, origin, dest, dDate, rDate)$;
$Inform(cust)$

endProc

where $Inform(cust)$ is a primitive action and $Register$ is a fluent in the language of situation calculus; $BookTransport(cust, origin, dest, dDate, rDate)$ and $BookAccomodations(cust, origin, dest, dDate, rDate)$ can be seen as call to other GOLOG procedures.

On the other hand, GOLOG gives to programs a logical semantics in extended situation calculus, thus makes possible to express program properties (like correctness or termination) and to reason about them by a theorem prover. In particular, being based on a formal theory of actions, the language allows to reason about program execution and to consider the effects of different courses of actions before committing to a particular behaviour. Given a domain description *Axioms* - which is defined as a collection of axioms including a set of domain-dependent axioms describing precondition and effects of atomic actions as well as the initial situation, plus a set of foundational domain-independent axioms of the situation calculus- executing a GOLOG program δ in a given initial situation S_0 amounts to establish the following entailment:

Program Execution

$$Axioms \models (\exists s) Do(\delta, S_0, s). \tag{3.1}$$

Notice that, like in Prolog, GOLOG programs are executed for their side effects, i.e. to obtain bindings for existentially quantified variables. A successful execution of the program, i.e. a successful proof, returns a binding for s: $s = do(a_n, ...(do(a_1, S_0)))$. $a_1, \ldots, a_n$ represents an execution trace of the program δ for the given initial situation. Intuitively by the above entailment we must find a legal sequence of actions (each action is executed in a context where its precondition are satisfied) which is a possible execution of the program δ starting from the situation S_0.

The query 3.1 can be extended by an additional condition on the final state ($\phi(s)$). This additional condition expresses a special case of the classical *planning task* and it can be very practical in many applications: the program definition constrains the search space of reachable situations in which to look for a legal sequence for achieving ϕ.

Various extensions of GOLOG have been developed. CONGOLOG incorporates concurrency, handling concurrent processes with different priorities, high-level interrupts [20, 21]. With IndiGolog programs can be executed incrementally to allow for interleaved action, planning, sensing, and exogenous events.

Original formulation of GOLOG did not deal with programs containing sensing actions, providing to the agent fresh knowledge to be used for deciding how to act. Such actions are crucial for allowing the agent to deal with incomplete knowledge about the domain or to monitor the value of fluents in domains where exogenous actions that are not under the agent's control might occur. Handling sensing actions requires to introduce an epistemic level for modeling agent's knowledge, which is missing in GOLOG. Moreover, in presence of sensing actions the planning task as expressed by 3.1 is no longer adequate and a more complex notion of plan than given by the classical view of plans as mere sequences of actions is required.

Levesque's planning theory builds on the top of an action theory based on classical situation calculus, suitably extended to handle sensing actions[61]. A new language for defining plans as robot programs is introduced. Robot programs may contain conditionals and loops, a sequence of actions being merely a special case. They may contain sensing actions as ordinary actions

and, similarly to [45, 12], it is possible to formally prove the correctness of a complex program respect to a given goal state. The planning task is specified as the problem to find a *robot program* that achieves a goal state when executed in a certain initial state, no matter how the sensing turns out. However, the paper does not suggest how to *automatically generate* such correct robot plans.

3.6.2 DyLOG

As GOLOG, the language DyLOG, that we are going to briefly review, is designed for specifying agents behaviour and for modeling dynamic systems. As a main difference, it arises fully inside the logic programming paradigm: while GOLOG programs are defined by procedural statements in an Algol-like language, DyLOG programs are defined by sets of Horn-like rules, and a SLD-style proof procedure is introduced, which executes programs as proves theorems.

DyLOG DyLOG is a high-level logic programming language for modeling rational agents. It is based on a modal theory of actions and mental attitudes where *action modalities* are used for representing primitive (see section 3.2.2) and complex *actions*, while *belief modalities* model the agent's internal state [8]. It accounts both for atomic and complex actions, or procedures, for specifying the agent behaviors. Atomic actions are either world actions, affecting the world, or mental actions, i.e. sensing and communicative actions which only affect the agent beliefs. Complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like rewrite rules (which are interpreted as axioms of the logic - grammar logics) and by making use of action operators from dynamic logic, like sequence ";", test "?" and non-deterministic choice "∪". Formally complex actions are defined by a set of axioms of the form $\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi$, where p_0 is the procedure name the p_i's can be i's primitive acts, sensing actions for modeling information reception, test actions (actions of the form $Fs?$, where Fs is conjunction of belief formulas) or procedure names.

A DyLOG agent can be provided with a *communication kit* that specifies of its communicative behavior [3], defined in terms of interaction protocols, i.e. conversation policies that build on FIPA-like speech acts. The communication theory is viewed as a homogeneous component of the general agent theory, as both conversational policies, that guide the agent's communicative behavior, and other policies defining the agent's complex behavior are represented by procedures definitions (procedure axioms).

The action theory allows to cope with the problem of reasoning about complex actions with incomplete knowledge and in particular to address the temporal projection and planning problem. Intuitively DyLOG allows to specify the behavior of a rational agent that reasons about its own behavior, chooses a course of actions conditioned on its mental state and can use sensors and communication for obtaining fresh knowledge. In this spirit it has already been used with success for agent programming, in implementing web applications. In [5] an application has been developed where a *virtual tutor* helps students to build personalized study curricula, based on the description of courses viewed as actions. Recent works [3, 2, 4] focussed on using reasoning about actions techniques supported by DyLOG for customizing the composition of services, based on a semantic description of the services. In particular the ability of reasoning about interaction protocols has been exploited for customizing service selection and composition w.r.t. to the user's constraints.

In DyLOG the agent behavior is described by a *domain description*, which includes, besides a specification of the agents initial beliefs, a description of the agent behavior plus a *communication kit* (denoted by CKit^{ag_i}), that encodes its *communicative behavior*. Communication is

supported both at the level of *primitive speech acts* and at the level of *interaction protocols*. Thus, the communication kit of an agent ag_i is defined as a triple $(\Pi_{\mathcal{C}}, \Pi_{\mathcal{CP}}, \Pi_{\mathcal{S}get})$: $\Pi_{\mathcal{C}}$ is a set of laws defining precondition and effects of the agent speech acts; $\Pi_{\mathcal{CP}}$ is a set of procedure axioms, specifying a set of interaction protocols, and can be intended as a library of *conversation policies*, that the agent follows when interacting with others; $\Pi_{\mathcal{S}get}$ is a set of sensing axioms for acquiring information by messages reception.

The following is an example of DyLOG procedure expressing a conversation policy that a customer service (agent C) could follow in order to interact with a restaurant booking web service. The subscripts next to the procedure names are a writing convention for representing the role that the agent plays; so, for instance, Q stands for querier, and C for customer.

(a) $\langle \mathsf{reserv\_rest}_C(Self, Service, Time)\rangle\varphi \subset$
$\langle \mathsf{yes\_no\_query}_Q(Self, Service, available(Time))$;
$\mathcal{B}^{Self} available(Time)?$;
$\mathsf{get\_info}(Self, Service, reservation(Time))$;
$\mathsf{get\_info}(Self, Service, cinema\_promo)$;
$\mathsf{get\_info}(Self, Service, ft\_number)\rangle\varphi$

The customer asks if a table is available at a certain time, if so, the restaurant informs it that a reservation has been taken and that it gained a promotional free ticket for a cinema ($cinema\_promo$), whose code number (ft_number) is returned. The question mark amounts to check the value of a fluent in the current state; the semicolon is the sequencing operator of two actions.
$\mathsf{yes\_no\_query}_Q(Self, Service, available(Time))$ is a procedure defining a conversation policy for asking information on the truth of a given fluent. It builds on the top of FIPA-like performative as $inform$ and $queryIf$ modelled as primitive actions in the modal action theory. get_info's are special sensing actions for modeling the reception of messages.

DyLOG allows reasoning about agents' behavior, by supporting reasoning techniques for proving 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, it can be intended as 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?".

Formally, given a DyLOG domain description Π_{ag_i} containing a CKit^{ag_i} with the specifications of the interaction protocols and of the relevant speech acts, a *planning* activity can be triggered by *existential queries* of the form $\langle p_1\rangle\langle p_2\rangle \ldots \langle p_m\rangle Fs$, where each p_k $(k = 1, \ldots, m)$ may be an atomic or complex action executed by the agent, or a sensing action. Checking if the query succeeds corresponds to answering to the question "is there an execution of p_1, ..., p_m leading to a state where the conjunction of belief formulas Fs holds for agent ag_i?". Such an execution is a *plan* to bring about Fs. As in GOLOG the procedure definition constrains the search space. Actions in the plan can be action performed by ag_i or special actions that can be read as the *assumption* that a certain input will be received as result of sensing.

The ability of making assumptions about the outcome of sensing is necessary in order to actually build the plan. Depending on the task that one has to execute, it may alternatively be necessary to take into account all of the possible all the possible sensing outcome or just to find one of them for which the goal is achieved. In the former case, the extracted plan will be *conditional*, and for each sensing action it will generally contain many branches as the possible sensing outcomes. Each path in the resulting tree is a linear plan that brings about Fs. In the latter case, instead, the plan is *linear*. A goal-directed proof procedure has been developed that

implements such kinds of agent's reasoning and planning, and allows to automatically extract linear or conditional plans from DyLOG procedures [8, 3].

3.6.3 IMPACT

IMPACT Impact [65, 59] is an international research project led by the University of Maryland. The principal goal of this project is to develop both a theory as well as a software implementation that facilitates the creation, deployment, interaction, and collaborative aspects of software agents in a heterogeneous, distributed environment.

In IMPACT system all agents have the same architecture and hence the same components, but the content of these components can be different, leading to different behaviors and capabilities offered by different agents. The system provides the possibility to select data structures that best suit the application functions desired by users of the application they are building, thus "agentizing" pieces of code of arbitrary software programs. Finally the agent is built on the top of it.

The first step is to have an abstract definition of what that body of code looks like. The system need a specification of the data types or data structures that the agent manipulates, with associate a domain which is the space of objects of that type. The above set of data structures is manipulated by a set of functions, constituting the *application programmer interface* or API of the package on top of which the agent is being built, that are callable by external programs.

Once defined the data structures, the data types and the functions manipulating them, is possible to use a unified query language to query them. If p is the name of a package, and f is an n–ary function defined in that package, then $p : f(a_1, \ldots, a_n)$ is a *code call*. It can be read as "execute function f as defined in package p on the stated list of arguments". An *atomic code call* condition is an expression of the form `in(X,p:f(`a_1`, ... ,` a_n`))` which succeeds if `X` is in the set of answers returned by the code call in question. Finally, a code call condition is a conjunction of atomic code call conditions and deconstruction and constraint operations[2].

In addition to the data types of the code that an agent is built on top of, IMPACT provides a special "messaging" package which may be "added on" to agents so that they are able to handle messaging. At any given point in time, the actual set of objects in the data structures (and message box) managed by the agent constitutes the state of the agent.

The agent has a set of actions that can change its state. Such actions may include reading a message from the message box, responding to a message, executing a request "as is", executing a modified request, cloning a copy of the agent and moving it to a remote host, updating the agent data structures, etc. Even doing nothing may be an action.

Every action has a precondition, a set of effects that describe how the agent state changes when the action is executed, and an execution script or method consisting of a body of physical code that implements the action. For instance, when a robotic agent executes the action `move((X,Y),(X1,Y1))`, not only must its state be updated to reflect the move, but an execution script to actually move the robot to its new location must be executed. Such an action may be stated as follows:

1. **Precondition**: `in((X;Y); robot data : location())`
2. **Add list**: `in((X1;Y1); robot data : location())`

[2] a deconstruction operation accesses the `x` field of a variable `V` ranging over records that have an `x` field. `V.x > 25` is a constraint which checks if this condition is true.

3. **Delete list**: `in((X;Y); robot data : location())`

The agent has an associated body of code implementing a notion of concurrency. Intuitively, a notion of concurrency takes a set of actions as input, and returns a single action (which "combines" the input actions together) as output. There are numerous possible notions of concurrency.

Each agent has a set of action and integrity constraints the states of the agent are expected to satisfy. The former are needed in order to prevent that some specific actions $\{a_1, \ldots, a_n\}$ are concurrently executed, the latter assure a sort of consistency.

Each agent has a set of rules called the Agent Program specifying the operating principles under which the agent is functioning. These rules describe the do's and dont's for the agent. They specify what the agent may do, what it must do, what it may not do, etc. The Agent Program uses deontic modalities to implement what the agent can and cannot do. 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 called action status atoms. These action status atoms are read (respectively) as $\alpha(\vec{t})$ is obligatory, permitted, forbidden, done, and the obligation to do $\alpha(\vec{t})$ is waived.

If A is an action status atom, then A and $\neg$A are called action status literals. An agent program is a finite set of rules of the form:

$$\mathrm{A} \longleftarrow \chi \wedge L_1 \wedge \ldots \wedge L_n$$

where A is an action status atom, χ is a code call condition, and $L_1 \wedge \ldots \wedge L_n$ are action status literals.

When the agent is initially constructed and deployed, all integrity constraints are satisfied by the agent's state so to ensure that whatever the agent does, it maintains consistency of the integrity constraints by never executing actions that force it to transition to an inconsistent state. An agent B can directly change an agent A's state only by sending it a message (and thus causing an update to the agent's mailbox). All other changes to agent A's state must be made by agent A, perhaps as a response to such a message from agent B. Everytime an agent A receives a message, its integrity constraints may get violated. The agent's job is to compute a set of actions to take which, if executed concurrently, satisfy the following conditions:

1. satisfies the action constraints;
2. leads to a new state (of the agent) that satisfies the integrity constraints;
3. satisfies all rules of the agent's program.

In fact, in this framework, a *status set* is a set of ground action status atoms that preserve such conditions, and also satisfy some consistency conditions.

An important point to note is that agents continuously respond to messages (state changes) by computing such a status set, and concurrently executing all the actions of the form $\mathbf{Do}\alpha$ in that status set. The entire implementation of the IMPACT system is built to efficiently and scalably support this need.

3.7 Reasoning about interaction on the semantic web

Recent years witnessed a rapid evolution of the concept of world-wide web, moving from an information providing web to one that provides "services". In this perspective the web is seen

as a platform that supports the execution of activities, not only carried on by human beings but in particular by software entities. The idea is to develop an infrastructure that allows the automatic retrieval of software devices –based on a machine-interpretable semantic description of what they will do–, their execution, and their automatic composition and interoperation, aimed at performing complex tasks. The challenge lays in the achievement of the capability of making a set of softwares, gathered on-the-fly according to the current goals of the user but developed independently, interoperate so to achieve a given goal. A lot of research is currently being carried on about these topics, and some proposals have been done, although none can yet be considered as "the solution".

One model that seems to fit particularly well the described scenario is to consider the services as agents, whose interoperation is based on communication. Indeed, the web service must follow some, possibly non-deterministic, procedure aimed at getting/supplying all the information necessary to perform its task or resulting from it. More precisely, talking about agents, a web service behavior can be expressed as a *conversation protocol*, which describes the communications that can occur with the other agents.

3.7.1 Interactions as communications

Communication and dialogue have intensively been studied in the context of formal theories of agency [25]. In particular, a great deal of 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, where it is fundamental to have a universally shared semantic.

Agent communication languages are complex structures because a communicative act must specify many kinds of information, such as its content and the kind of performative. The definition of formal semantics for individual communicative acts has been one of the major topics of research in this field. Most of the proposals are ultimately based on the philosophical theory of speech acts developed by Austin and Searle in the sixties. Following the basic insight of the speech act theory, communications are not just considered as transmitting information but as *actions* that, instead of modifying the external world, affect the mental states of the involved agents. As a consequence, individual speech act semantic has been given in terms of preconditions and effects on mental attitudes, as it is commonly done with action semantic, and standard techniques for reasoning about change have been exploited for proving conversation properties, planning communication with other agents and for answer selection. In this line, many approaches in the literature are based on variants of modal logic, in which mental attitudes, such as beliefs, goals and intentions, as well as communicative acts are represented by modalities [14, 39, 26].

Only recently the attention has been moved to formalize those aspects of communication that are related to the conversational context in which communicative acts occur [46]. The formalization of conversation policies adds a higher semantic level, which improves the interoperability of the various components (often separately developed) and simplifies the verification of compliance to the desired standards. In the area of agent languages based on logic, some examples of definition of protocols for guiding the agent communicative behavior can be found in [60, 3]. By working at the level of protocols, agents can more easily be seen as individuals, developed independently, on different platforms and with different approaches, a very attractive view in the applicative field of web applications and web services. For all these reasons we focus on a semantics of communication that supports the specification and reasoning about single

speech acts, as well as the specification and reasoning about speech acts in the context of a conversation protocol.

Instead of referring to a mentalistic approach as described above, some authors have proposed a social approach to agent communication [64, 35]. The reason is that the mental approach is not well suited to those cases in which the history of communications is observable but the internal states of the single agents are not. In the social approach communicative actions affect the "social state" of the system, which consists of social facts, like the *permissions* and the *commitments* of the agents, which are created and modified along the interaction. The dynamics of the system emerges from the interactions of the agents, which must respect these permissions and the commitments (if they are compliant with the protocol). 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.

3.7.2 Representing protocols of interaction

In this section we describe in more details the approach to protocol representation, that is based on the action metaphor. The reason is that by this approach it is possible to represent in a uniform way the communicative behavior and the non-communicative behavior of the agents.

The basic idea of this approach is that protocols are described by procedures built upon a predefined set of speech acts. Each speech act is an atomic communicative action, usually involving two agents. In the last years a lot of efforts have been devoted to the study of speech acts. One of the major results is due to FIPA, a non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents. The FIPA Agent Communication Language (FIPA ACL) defines not only the syntax of a set of speech acts, but also their semantics, which is but trivial.

For instance, in the specifications of FIPA ACL, we read that the *inform* speech act denotes a communication between a sending agent and a receiving agent, that indicates that the sending agent knows that some proposition is true, that it intends that the receiving agent also comes to believe that the proposition is true, and, that it does not already believe that the receiver has any knowledge of the truth of the proposition. The first two properties defined above are straightforward: the sending agent is sincere, and it has generated the intention that the receiver should know the proposition (for instance, because it has been asked). The last property is concerned with the semantic soundness of the act. If an agent already knows that some state of the world holds (in our case, that the receiver already knows the truth of the proposition), it cannot rationally adopt an intention to bring about that state of the world, i.e. it will not perform the inform act.

Note that the property is not as strong as it might appear. In fact, the sender is not required to establish whether the receiver knows the proposition. On the other hand, from the receiver's point of view, an inform message entitles it to believe that the sender believes the content of the message, and that the sender wishes the receiver to believe that proposition as well. Whether or not the receiver will actually believe the proposition is a function of the receiver's trust in the sincerity and reliability of the sender.

From this example we can notice that speech acts may have preconditions to their execution and effects, that are described in terms of what the involved agents know (or believe) about the object of the communication and about each other. Therefore, they can suitably be represented by means of agent programming languages for reasoning about actions and change. One such language, that is particularly interesting because it not only allows the definition of speech

acts but also of conversation protocols, is DyLOG, introduced in [7, 8] and set in a modal logic framework.

Representing interaction protocols in DyLOG

In DyLOG, the integration of a *communication theory* [55, 3] in the general agent theory is obtained by adding further axioms and laws to the agent domain description. In the following we will denote agents by ag_i or ag_j. Let us now briefly describe the language components that allow communication, which are: speech acts, get message actions, and protocols.

Speech Acts are atomic actions, described in terms of preconditions and effects on the agent mental state. They have the form $\mathsf{speech\_act}(sender, receiver, l)$, where *sender* and *receiver* are agents and l is either a fluent literal or a done fluent. Such actions can be seen as special mental actions, affecting both the sender's and the receiver's mental state. In our model we focused on the *internal representation*, that agents have of each speech act, by specifying ag_i's belief changes both when it is the sender and when it is the receiver. They are modelled by generalizing the action and precondition laws so to allow the representation of the effects of communications performed by other agents on ag_i mental state. Such a representation provides the capability of *reasoning about* conversation effects. Speech act specification is, then, twofold: one definition holds when the agent is the sender, the other when it is the receiver. In the first case, the precondition laws contain some *sincerity condition* that must hold in the agent mental state. When ag_i is the receiver, the action is *always* executable. Let us consider as an example the DyLOG representation of the FIPA ACL *inform* speech act, described a few paragraphs above:

a) $\Box(\mathcal{B}^{ag_i} l \wedge \mathcal{B}^{ag_i}\mathcal{U}^{ag_j} l \supset \langle \mathsf{inform}(ag_i, ag_j, l)\rangle\top)$
b) $\Box([\mathsf{inform}(ag_i, ag_j, l)]\mathcal{M}^{ag_i}\mathcal{B}^{ag_j} l)$
c) $\Box(\mathcal{B}^{ag_i}\mathcal{B}^{ag_j} authority(ag_i, l) \supset [\mathsf{inform}(ag_i, ag_j, l)]\mathcal{B}^{ag_i}\mathcal{B}^{ag_j} l)$
d) $\Box(\top \supset \langle \mathsf{inform}(ag_j, ag_i, l)\rangle\top)$
e) $\Box([\mathsf{inform}(ag_j, ag_i, l)]\mathcal{B}^{ag_i}\mathcal{B}^{ag_j} l)$
f) $\Box(\mathcal{B}^{ag_i} authority(ag_j, l) \supset [\mathsf{inform}(ag_j, ag_i, l)]\mathcal{B}^{ag_i} l)$
g) $\Box(\mathcal{M}^{ag_i} authority(ag_j, l) \supset [\mathsf{inform}(ag_j, ag_i, l)]\mathcal{M}^{ag_i} l)$

Clause (a) states that an inform act can be executed when the sender believes l and believes that the receiver does not know l. When ag_i is the sender it thinks possible that the receiver will adopt its belief, although it cannot be sure about it –autonomy assumption (b)–. If it believes that ag_j considers it a trusted *authority* about l, it is confident that the receiver will adopt its belief (c). When ag_i is the receiver, it believes that l is believed by the sender ag_j (e), but it adopts l as an own belief only if it thinks that the latter is a trusted authority (f)-(g).

Get Message Actions are used for *receiving* messages from other agents. They are modeled as a special kind of sensing actions, because from the agent perspective they correspond to queries for an external input, whose outcome is unpredictable. The main difference w.r.t. normal sensing actions is that they are defined by means of speech acts performed by the interlocutor. Formally, we use get_message actions defined by an axiom schema of the form:

$$[\mathsf{get\_message}(ag_i, ag_j, l)]\varphi \equiv [\bigcup_{\mathsf{speech\_act} \in \mathcal{C}_{\mathsf{get\_message}}} \mathsf{speech\_act}(ag_j, ag_i, l)]\varphi \qquad (3.2)$$

Intuitively, $\mathcal{C}_{\mathsf{get\_message}}$ is a finite set of speech acts, which are all the possible communications that ag_i expects from ag_j in the context of a given conversation. We do not associate to a

Figure 3.1: The AUML graph represents the communicative interactions occurring between the *querier* and the *informer* in the yes_no_query protocol.

get_message action a domain of mental fluents, but we calculate the information obtained by looking at the effects of the speech acts in $\mathcal{C}_{\mathsf{get\_message}}$ on ag_i's mental state.

Conversation protocols We suppose individual speech acts to take place in the context of predefined conversation protocols [46] that specify communication patterns. Each agent has a subjective perception of the communication with other agents, for this reason each protocol has as many procedural representations as the possible roles in the conversation. Let us consider, for instance the yes_no_query protocol reported in Fig. 3.1, a simplified version of the FIPA Query Interaction Protocol [26]. The protocol has two complementary views, one to be followed for making a query ($\mathsf{yes\_no\_query}_Q$) and one for responding ($\mathsf{yes\_no\_query}_I$). In the following get_answer and get_start definitions are instances of the get_message axiom.

$$\langle \mathsf{yes\_no\_query}_Q(Self, Other, Fluent)\rangle\varphi \subset \langle \mathsf{queryIf}(Self, Other, Fluent); \mathsf{get\_answer}(Self, Other, Fluent)\rangle\varphi$$

$$[\mathsf{get\_answer}(Self, Other, Fluent)]\varphi \equiv [\mathsf{inform}(Other, Self, Fluent) \cup \mathsf{inform}(Other, Self, \neg Fluent) \cup \mathsf{refuseInform}(Other, Self, Fluent)]\varphi$$

Intuitively, the right hand side of get_answer represents all the possible answers expected by agent $Self$ from agent $Other$ about $Fluent$, in the context of a conversation ruled by the $\mathsf{yes\_no\_query}_Q$ protocol.

$$\langle \mathsf{yes\_no\_query}_I(Self, Other, Fluent)\rangle\varphi \subset \langle \mathsf{get\_start}(Self, Other, Fluent); \mathcal{B}^{Self} Fluent?; \mathsf{inform}(Self, Other, Fluent)\rangle\varphi$$
$$\langle \mathsf{yes\_no\_query}_I(Self, Other, Fluent)\rangle\varphi \subset \langle \mathsf{get\_start}(Self, Other, Fluent); \mathcal{B}^{Self} \neg Fluent?; \mathsf{inform}(Self, Other, \neg Fluent)\rangle\varphi$$
$$\langle \mathsf{yes\_no\_query}_I(Self, Other, Fluent)\rangle\varphi \subset \langle \mathsf{get\_start}(Self, Other, Fluent); \mathcal{U}^{Self} Fluent?; \mathsf{refuseInform}(Self, Other, Fluent)\rangle\varphi$$

The $\mathsf{yes\_no\_query}_I$ protocol specifies the behavior of the agent $Self$, that waits a query from $Other$; afterwards, it replies according to its beliefs on the query subject. get_start is a get_message action ruled by the following axiom:

$$[\mathsf{get\_start}(Self, Other, Fluent)]\varphi \equiv [\mathsf{queryIf}(Other, Self, Fluent)]\varphi$$

We can define the **communication kit** of an agent ag_i, CKit^{ag_i}, as the triple $(\Pi_{\mathcal{C}}, \Pi_{\mathcal{CP}}, \Pi_{\mathcal{S}get})$, where $\Pi_{\mathcal{C}}$ is the set of simple action laws defining ag_i's primitive speech acts, $\Pi_{\mathcal{S}get}$ is a set of axioms for ag_i's get_message actions and $\Pi_{\mathcal{CP}}$ is the set of procedure axioms specifying the ag_i's conversation protocols. Thus a **domain Description** for an agent ag_i is defined as a triple $(\Pi, \mathsf{CKit}^{ag_i}, S_0)$, where CKit^{ag_i} is a communication kit, S_0 is ag_i's initial set of belief fluents, and Π describes the non-communicative behavior of the agent. In particular, Π is a tuple

$(\Pi_{\mathcal{A}}, \Pi_{\mathcal{S}}, \Pi_{\mathcal{P}})$, where $\Pi_{\mathcal{A}}$ is the set of ag_i's world action (actions that affect the environment of the agent) and their precondition laws, $\Pi_{\mathcal{S}}$ is a set of axioms for ag_i's sensing actions, $\Pi_{\mathcal{P}}$ a set of procedure axioms (complex actions), as well used for affecting the world.

By exploiting nested beliefs a *subjective* representation of conversation protocols has been taken, in which an agent makes rational assumptions on its interlocutor's state of mind. Notice that, since we are only interested in reasoning about the dynamics of the *local* mental state, our approach differs from other logic-based approaches to communication in multi-agent systems, as the one taken in Congolog [63], where communicative actions affect the global state of a multi-agent system.

3.7.3 Reasoning about communication and protocols

Reasoning about conversation protocols is a means for proving properties of the communication in a group of agents, verifying if it satisfies some properties of interest. In the case in which communication is expressed as the effect of a set of communicative actions (speech acts), reasoning means to wonder about the changes occurring to the mental state of an agent –or of the set of agents– involved in the communication. By doing this, it is possible to understand the effectiveness of a protocol with respect to a task of interest for the agent that might initiate the conversation. for instance, the agent might decide whether to select a certain service depending on what the interaction allows or does not allow to communicate during the interaction. Another interesting case is verify if the composition of a set of protocols, each followed by a different agent, will allow the achievement of a desired goal.

Reasoning about communications and protocols in DyLOG

As we have seen, a communication kit, integrated in DyLOG, allows to describe the communicative behavior on an agent in an explicit and high-level way. Agents can reason about these descriptions and, thus, about the interactions that they are *going to* enact. One useful task is to prove if there is a possible execution of the protocol, after which a set of beliefs of interest (or goal) will be true in the agent mental state. Notice that such a form of reasoning implies making assumptions about the mental state of *other* agents, those with which the interaction will take place.

Let us recall that, given a DyLOG domain description, it is possible to reason about it and formalize the *temporal projection* and the *planning* problem by means of existential queries of form:

$$\langle p_1 \rangle \langle p_2 \rangle \dots \langle p_m \rangle Fs \tag{3.3}$$

Each p_k, $k = 1, \dots, m$ in (3.3) may be an (atomic or complex) action executed by ag_i or an external speech act, that belongs to CKit^{ag_i} (by *external* we denote a speech act in which our agent plays the role of the receiver). Checking if a query of form (3.3) succeeds corresponds to answering to the question "is there an execution trace of $p_1, \dots, p_m$ leading to a state where the conjunction of belief fluents Fs holds for ag_i?". Such an execution trace is a *plan* to bring about Fs. The procedure definition constrains the search space.

In presence of communication, the problem of reasoning about *conversation protocols* is faced (a conversation is a sequence of speech acts). Indeed, in the case in which the p_i's are conversation protocols, by answering to the query (3.3) we find a conversation after which some desired condition Fs holds. This kind of reasoning is extremely useful in the application framework of service interoperation over the web. Given a service communication protocol,

it makes in fact possible to find a specific interaction sequence, which on a hand allows the achievement of the desired goal, while on the other, it is an instance of the conversation protocol made public by a service of interest or resulting from the composition of the protocols followed by a group of services that should cooperate. Conversations usually contain actions that allow the reception of messages from other agents, whose content is unknown at planning time. Nevertheless, the existence itself of a protocol allows the agent to make assumptions on such messages, which must range in the set of the possible answers foreseen by it.

Depending on the application, it might be reasonable to choose an approach that allows the agent to extract a conditional plan that leads to the goal independently from the answers of the interlocutor, as done in [3]. Alternatively, it might be useful to find a linear plan that leads to the goal, given that some assumptions on the received answers hold. This weaker approach does not guarantee that at *execution time* the services will attain to the planned conversation, but it allows us to find a feasible solution when a conditional plan cannot be found. For instance, consider a service for booking cinema tickets: before the actual interaction takes place it is impossible to know if a sufficient number of seats is actually free, however, the decision of whether beginning an interaction with that service does not depend on knowing in advance that the goal will be reached but rather on the fact that the interaction will occur in a way that satisfies the requirements (for instance, to be sure that the service will not ask for a telephone number). Actually, if no seat is available the goal of making a reservation will fail. The real advantage is that the information contained in the protocol is sufficient to exclude a number of alternative services (or possible composition in the case of service interoperation) that, anyhow, would never satisfy the goal.

DyLOG proof procedure is a natural evolution of [7, 8] and is described in [55]; it is goal-directed and based on negation as failure (NAF). NAF is used to deal with the persistency problem for verifying that the complement of a mental fluent is not true in the state resulting from an action execution, while in the modal theory we adopted an abductive characterization. The proof procedure allows agents to find *linear* plans for reaching a goal from an incompletely specified initial state. The soundness can be proved under the assumption of e-consistency, i.e. for any action the set of its effects is consistent [23]. The extracted plans always lead to a state in which the goal condition *Fs* holds.

3.7.4 Semantic Web

The semantic web effort is aimed at transforming the web, currently constructed so to be browsed, sought, and used by humans, in a platform that allows the automatic retrieval and invocation of resources by machines themselves. To this aim, the first necessary step is to enrich the representation of web resources by means of semantic information represented so to be machine-interpretable. A second aspect of the transformation to which the web is undergoing, is that resources more and more often are functionalities, software or hardware devices, such as weather forecast services or web-cams. Hardware/software devices accessible via the web are called "web services". In this context, there is is a growing need of allowing the users to perform tasks that require the retrieval of services based on the description of their function and also their composition, aimed at executing complex tasks. For instance, organizing a trip might require the use of services for booking train tickets, reserving a room at a hotel and checking bus schedules. These tasks are currently executed by the user him/herself, but it is desirable to develop techniques for retrieving and composing services in an automatic way. One possibility is to use the techniques, developed in Artificial Intelligence, for performing symbolic

manipulation (e.g. reasoning techniques). To this aim, it is necessary to have access to a high-level description of the functionalities of the service and, in particular, to how it works. In the literature it is already possible to find a few proposals of languages for web service description and for the description of their interactions, i.e. OWL-S [54] and WSDL/BPEL4WS [17, 13].

Languages for Web Services representation OWL-S, BPEL4WS

As the huge amount of information on the web urged the development of standard languages for representing the semantics behind the HTML (e.g. RDF [57], OWL [53]), recently some attempt to standardize the description of web services has been carried on (OWL-S –previously DAML-S– [54, 18], WSDL [68], BPEL4WS). The use of standard descriptions is aimed at allowing the automatic discovery of web services, their automatic execution and monitoring, and (the task we will focus on in this paper) *automatic composition*.

While the BPEL/WSDL initiative is mainly carried on by the commercial world, with the aim of standardizing registration, look-up mechanisms and interoperability, OWL-S is more concerned with providing greater expressiveness to service description in a way that can be reasoned about [15]. In particular, a service description has three conceptual levels: the *profile*, used for advertising and discovery, the *process model*, that describes how a service works, and the *grounding*, that describes how an agent can access the service.

More precisely, three parties are involved in a web service transaction: the service requester, the provider, and some infrastructure components. The service requester seeks a service provider to which delegate the execution of a task. In an open environment such as the Internet, the requester may not know ahead of time of the existence of the provider; in order to find one, it relies on infrastructure components, that act like registries. Within the OWL-S framework, the service profile provides a way to describe the services offered by the providers, and the services needed by the requesters. An OWL-S profile describes a service as a function of three types of information: which organization provides the service (contact information), what function the service computes (the performed transformation described in terms of required inputs and produced outputs, plus conditions on the environment that must be satisfied in order for the transformation to occur and produce results), and the characteristics of the service (the categorization of the service w.r.t. to a reference ontology).

A service profile provides a concise description of the service to a registry, and ideally its only purpose is advertisement. Once the service has been selected, however, the requester will control the interaction with the service by means of the process model. In particular, the process model describes a service as an atomic, a simple or a composite process in a way inspired by the language Golog and its extensions [44, 28, 48]. Atomic processes are directly invocable, they have no subprocesses, and execute in a single step. Simple processes, instead, conceived of as elements of abstraction; for instance they can be considered as a simplified view of a composite process. Composite processes are decomposable into simpler processes. Their structure can be specified by using the following control constructs: sequence, split, split + join, choice, unordered, condition, if-then-else, iterate, repeat-while, and repeat-until. Such a decomposition normally shows, among other things, how the various inputs of the process are accepted by particular subprocesses, and how its various outputs are returned by particular subprocesses.

In this perspective, a wide variety of agent technologies based upon the *action metaphor* can be used. In fact, we can view a service as an action (atomic or complex) with preconditions and effects, that modifies the state of the world and the state of agents that work in the world.

The process model can, then, be viewed as the description of such an action; therefore, it is possible to design agents, which apply techniques for reasoning about actions and change to web service process models for producing new, composite, and customized services.

The task of web service composition in the OWL-S context has been faced by some researchers. The most relevant work has been carried on by McIlraith et al. [15].

The basic idea is that services correspond to *atomic* actions, whose composition is based on their preconditions and effects. In particular, the precondition and effect, input and output lists are flat; no relation among them can be expressed, so it is impossible to understand if a service can follows an interaction protocol that allows various interactions. Indeed, the advantage of working at the protocols level is that by reasoning about protocols agents can personalize the interaction by selecting a course that satisfies user- (or service-) given requirements. This process can be started before the actual interaction takes place and can be exploited for web service composition as well as for web service search.

An alternative language to OWL-S is BPEL4WS (Business Process Execution Language for Web Services [13]), a notation for specifying business process behavior based on Web Services. In this approach, a business process is either executable, and in this case it models the actual behavior of a participant in a business interaction, or it is a so called business protocol, i.e. a description of the mutually visible message exchange behavior of each of the involved parties, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. BPEL4WS was born to overcome the limits of WSDL, the W3C standard language for web service description. The interaction model supplied by WSDL is essentially a stateless model of synchronous or asynchronous interactions. However, models for business interactions typically assume sequences of (either synchronous or asynchronous) message exchanges, within stateful, long-running interactions involving two or more parties. To define such interactions, a formal description of the used message exchange protocols is needed, in which the mutually visible message exchange behavior of each of the parties is clearly specified, without revealing the internal implementation of the involved processes. BPEL4WS separates "transparent" data, i.e. data relevant to public aspects, as opposed to "opaque" data, that internal/private functions use. Transparent data directly affects the public business protocol, whereas opaque data is significant to back-end systems and affects the protocol only by creating nondeterminism. Going back to the ticket booking example, the (non)availability of a sufficient number of seats would be opaque data, and cause nondeterminism. BPEL4WS explicitly allows the use of nondeterministic data values to make it possible to capture the essence of public behavior while hiding private aspects. The language is layered on top of several XML specifications (WSDL 1.1, XML Schema 1.0, and XPath1.0) but makes no use of semantic information; in particular, WSDL messages and XML Schema type definitions provide the data model used by BPEL4WS processes. The language is quite rich and, for each received message, it allows the definition of activity to execute by means of the following operators: receive, reply, invoke, assign, throw, terminate, wait, empty, sequence, switch, while, pick, flow, scope, compensate.

Research on the use of planning techniques for composing web services described by means of BPEL4WS has been carried on by some researchers, such as the one by Pistore, Traverso and Bertoli.

3.8 Action Languages and Semantic Web Services

3.8.1 A simple scenario

In this section we will define a simple scenario aimed at showing the advantage of expressing and reasoning about the interaction protocol followed by web services.

Let us consider a software agent (we will refer to it as *pa*) whose task is to crawl the internet for executing specific requests of a given user; indeed, *pa* is a user personal assistant. Let us suppose that *pa* current task is to book a ticket at a cinema where a given movie is shown. In a web service context, it will have to look for a provider of a cinema booking service by consulting a registry, and interact with it accordingly, supplying the requested information. As a further condition, let us imagine that the user requested the personal assistant not to use his credit card number in the upcoming transaction. Suppose also that two cinema booking services are

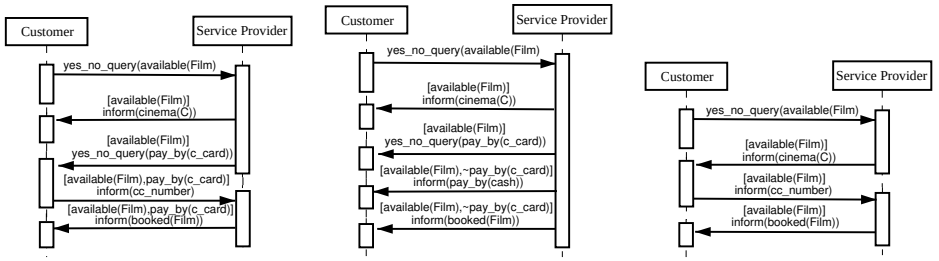


Figure 3.2: The three AUML graphs [52] represent the communicative interactions occurring between the customer (*pa*) and the provider; (a) and (b) are followed by *click_ticket*, (c) is followed by *all_cinema*. Formulas among square brackets represent conditions on the execution of the speech act.

available, called *click_ticket* and *all_cinema* respectively, that apply two different interaction protocols, one permitting both to book a ticket to be paid later by cash (Fig. 3.2 (a)) and to buy it by credit card (Fig. 3.2 (b)), the other allowing only ticket purchase by credit card (Fig. 3.2 (c)). These descriptions would induce a human assistant to choose *click_ticket*, selecting the option to pay cash; this choice can be done because we can reason about the consequences of communicative acts and procedures.

3.8.2 Reasoning about interaction for selecting and composing services in DyLOG

Web service selection

The problem that the personal assistant *pa* has in the Web service scenario outlined above can be read as an example of web service selection. In this section we show how it can be naturally turned into a planning problem in presence of communication, as the one treated by DyLOG [3, 2]. In fact the question *pa* tries to answer is: "is there some possible conversation, that is an instance of the protocol followed by the Web service provider and satisfies all the conditions posed by the user (e.g. at the and of the interaction the service must not know the user's credit card number)?". In a way, *pa* wonders if it is possible to personalize the interaction with its

interlocutor so to achieve certain goals. Let us take a DyLOG domain description containing the description of the $\mathsf{get\_ticket\_1}_C$ protocol reported above, suppose that *pa* knows the credit card number (*cc_number*) of the user but it is requested not to use it, and consider the query:

$$\langle \mathsf{get\_ticket\_1}_C(pa, click\_ticket, akira)\rangle \mathcal{B}^{pa} \neg \mathcal{B}^{click\_ticket} cc\_number$$

that amounts to determine if there is a conversation between *pa* and *click_ticket* about the movie *akira*, that is an instance of the conversation protocol $\mathsf{get\_ticket\_1}_C$, after which the service does not know the credit card number of the user.

Agent *pa* works on the behalf of a user, thus it knows the user's credit card number ($B^{pa} cc\_number$) and his desire not to use it in the current transaction ($\neg \mathcal{B}^{pa} pay\_by(c\_card)$). It also believes to be an authority about the form of payment and about the user's credit card number and that *click_ticket* is an authority about cinema and tickets. This is represented by the beliefs:
$\mathcal{B}^{pa} authority(pa, cc\_number)$ and $\mathcal{B}^{pa} authority(click\_ticket, booked(akira))$. The initial mental state will also contain the fact that *pa* believes that no ticket for *akira* has been booked yet, $\mathcal{B}^{pa} \neg booked(akira)$, and some hypothesis on the interlocutor's mental state, e.g. the belief fluent $\mathcal{B}^{pa} \neg \mathcal{B}^{click\_ticket} cc\_number$, meaning that the web service does not already know the credit card number. Suppose, now, that the ticket is available; since *pa* mental state contains the belief $\neg \mathcal{B}^{pa} pay\_by(c\_card)$, when it reasons about the protocol execution, the test on $\mathcal{B}^{pa} pay\_by(c\_card)$? fails. Then clause (b) is to be followed, leading *pa* to be informed that it booked a ticket, $\mathcal{B}^{pa} booked(akira)$, which is supposed to be paid cash. No communication involves the belief $\mathcal{B}^{pa} \neg \mathcal{B}^{click\_ticket} cc\_number$, which persists from the initial state. Even when the ticket is not available or the movie is not known by the provider, the interaction ends without consequences on the fluent $\mathcal{B}^{pa} \neg \mathcal{B}^{click\_ticket} cc\_number$. The briefly described reasoning process leads to find an execution trace of $\mathsf{get\_ticket\_1}_C$, which corresponds to a *personalized conditional dialogue plan* between *pa* and the provider *click_ticket*, always leading to satisfy the user goal of not giving the credit card number.

One important observation: the fact that it is possible to plan a conversation that leads to the fulfillment of some goal does not imply that by its execution the goal will actually be achieved. For instance, think to the case in which the user got a free ticket but no free seat remains at the selected cinema. The fact that the cinema is fully booked is an information that will be known only at execution time. The planned conversation is, actually, a linear plan that leads to the goal given that some assumptions about the possible answers of the interlocutor are respected during the interaction. If such assumptions are not satisfied at execution time, the plan fails. It would be interesting to continue the presented work by integrating in this approach to service composition a mechanism for dealing with failure and replanning. This form of reasoning is necessary in order to arrive to real applications (e.g. a recommendation system) and it could take into account also preference criteria explicitly expressed by the user. Such criteria could be used to relax some of the constraints in case no plan can be found or when a plan execution fails.

Web service composition

Let us now consider the case of a user who wants to spend an evening out by going both to a restaurant and then to a cinema. He wants to make a reservation at both places and he is a little restrictive about the possible alternatives. He wants to see a specific movie (e.g. Nausicaa) and he wishes to benefit of some promotion on the cinema ticket but he is not eager

to communicate his credit card number on the internet. If, on one hand, searching for a cinema or a restaurant reservation service is a task that can be accomplished on the basis of a set of characteristic keywords, stored in a registry system used for advertisement, the other kinds of condition (look for promotions, do not use credit card) can be verified only by reasoning about the way in which the web service operates and, in particular, about the interaction protocol that it follows. To complete the example, suppose that two restaurants and two cinemas are available, but only *restaurant1* takes part to a promotion campaign, by which it gives to each customer, who made a reservation by the internet service, a *free ticket* for a movie. On the side of cinemas, suppose that *cinema2* accepts reservations but no free ticket, whereas *cinema1* accepts to make reservations by using promotional tickets or by using the credit card.

We imagine the search and composition process as divided in two steps. The first step (not described in this work) is *keyword-based* and is aimed at restricting the attention to a little set of services, extracted from a registry system. The second step is a further selection based on *reasoning* which is realized by using DyLOG [4]. During this step the agent personalizes the interaction with the services according to the requests of the user and dismisses services that do not fit. To this aim, the agent reasons about the procedure *comp_services* (a possible implementation of it is reported below), that sketches the general composition-by-sequencing of a set of services, based on the interaction protocols ($service(TypeService, Name, Protocol)$), that we suppose explicitly given in the service descriptions identified by the first step.

$\langle comp\_services([\,])\rangle\varphi \supset true$
$\langle comp\_services([[TypeService, Name, Data]|Services])\rangle\varphi \supset$
$\quad \langle \mathcal{B}^{pa} service(TypeService, Name, Protocol)$;
$\quad Protocol(pa, Name, Data)$;
$\quad comp\_services(Services)\rangle\varphi$

Intuitively, *comp_services* builds the sequence of protocols to apply for interacting with a set of services, so that it will be possible to reason about the whole. The presented implementation is quite simple but is sufficient as an example and generally it could be any Prolog-like procedure. Before explaining the kind of reasoning that can be applied, let us describe the *protocols*, that are followed by the web services of our example. Such protocols allow the interaction of two agents, so each of them has two complementary views: the view of the web service and the view of the customer, i.e. *pa*. In the following we will report (written in DyLOG) the view that *pa* has of the protocols.

(a) $\langle \mathsf{reserv\_rest}_C(Self, WebS, Time)\rangle\varphi \subset$
$\quad \langle \mathsf{yes\_no\_query}_Q(Self, WebS, available(Time))$;
$\quad\quad \mathcal{B}^{Self} available(Time)?$;
$\quad\quad \mathsf{get\_info}(Self, WebS, reservation(Time))$;
$\quad\quad \mathsf{get\_info}(Self, WebS, cinema\_promo)$;
$\quad\quad \mathsf{get\_info}(Self, WebS, ft\_number)\rangle\varphi$

(b) $\langle \mathsf{reserv\_rest\_2}_C(Self, WebS, Time)\rangle\varphi \subset$
$\quad \langle \mathsf{yes\_no\_query}_Q(Self, WebS, available(Time))$;
$\quad\quad \mathcal{B}^{Self} available(Time)?$;
$\quad\quad \mathsf{get\_info}(Self, WebS, reservation(Time))\rangle\varphi$

(c) $[\mathsf{get\_info}(Self, WebS, Fluent)]\varphi \subset [\mathsf{inform}(WebS, Self, Fluent)]\varphi$

Procedures (a) and (b) describe the customer-view of the protocols followed by the two considered restaurants. The customer asks if a table is available at a certain time, if so, the service informs the customer that a reservation has been taken. The first restaurant also informs the customer that it gained a promotional free ticket for a cinema ($cinema\_promo$) and it returns a code number (ft_number).

(d) $\langle \mathsf{reserv\_cinema}_C(Self, WebS, Film)\rangle\varphi \subset$
$\langle \mathsf{yes\_no\_query}_Q(Self, WebS, available(Film))$;
$\mathcal{B}^{Self} available(Film)?$;
$\mathsf{yes\_no\_query}_I(Self, WebS, cinema\_promo)$;
$\neg\mathcal{B}^{Self} cinema\_promo?$;
$\mathsf{yes\_no\_query}_I(Self, WebS, pay\_by(c\_card))$;
$\mathcal{B}^{Self} pay\_by(c\_card)?$;
$\mathsf{inform}(Self, WebS, cc\_number)$;
$\mathsf{get\_info}(Self, WebS, reservation(Film))\rangle\varphi$

(e) $\langle \mathsf{reserv\_cinema}_C(Self, WebS, Film)\rangle\varphi \subset$
$\langle \mathsf{yes\_no\_query}_Q(Self, WebS, available(Film))$;
$\mathcal{B}^{Self} available(Film)?$;
$\mathsf{yes\_no\_query}_I(Self, WebS, cinema\_promo)$;
$\mathcal{B}^{Self} cinema\_promo?$;
$\mathsf{inform}(Self, WebS, ft\_number)$;
$\mathsf{get\_info}(Self, WebS, reservation(Film))\rangle\varphi$

(f) $\langle \mathsf{reserv\_cinema\_2}_C(Self, WebS, Film)\rangle\varphi \subset$
$\langle \mathsf{yes\_no\_query}_Q(Self, WebS, available(Film))$;
$\mathcal{B}^{Self} available(Film)?$;
$\mathsf{get\_info}(Self, WebS, pay\_by(cash))$;
$\mathsf{get\_info}(Self, WebS, reservation(Film))\rangle\varphi$

Clauses (d) and (e) are the protocol followed by *cinema1*, (f) is the protocol followed by *cinema2*. Supposing that the desired movie is available, *cinema1* alternatively accepts credit card payments (d) or promotional tickets (e). *Cinema2*, instead, does not take part to the promotion campaign.

Let us now consider the query:

$\langle comp\_services([[restaurant, R, dinner], [cinema, CIN, nausicaa]])\rangle$
$(\mathcal{B}^{pa} cinema\_promo \wedge \mathcal{B}^{pa} reservation(dinner) \wedge$
$\mathcal{B}^{pa} reservation(nausicaa) \wedge \mathcal{B}^{pa}\neg\mathcal{B}^{CIN} cc\_number \wedge \mathcal{B}^{pa}\mathcal{B}^{CIN} ft\_number)$

that amounts to determine if it is possible to compose the interaction with a restaurant web service and a cinema web service, so to reserve a table for dinner ($\mathcal{B}^{pa} reservation(dinner)$) and to book a ticket for the movie Nausicaa ($\mathcal{B}^{pa} reservation(nausicaa)$), exploiting a promotion ($\mathcal{B}^{pa} cinema\_promo$). The user also specifies that no credit card is to be used ($\mathcal{B}^{pa}\neg\mathcal{B}^{CIN} cc\_number$), instead the obtained free ticket is to be spent ($\mathcal{B}^{pa}\mathcal{B}^{CIN} ft\_number$), i.e. *pa* believes that after the conversation the chosen cinema will know the number of the ticket given by the selected restaurant but it will not know the user's credit card number. Let us suppose that *pa* has the following list of available services:

$\mathcal{B}^{pa} service(restaurant, restaurant1, reserv\_rest_C)$
$\mathcal{B}^{pa} service(restaurant, restaurant2, reserv\_rest\_2_C)$
$\mathcal{B}^{pa} service(cinema, cinema1, reserv\_cinema_C)$
$\mathcal{B}^{pa} service(cinema, cinema2, reserv\_cinema\_2_C)$

then the query succeeds with answer R equal to $restaurant1$ and CIN equal to $cinema1$. This means that there is first a conversation between pa and $restaurant1$ and, then, a conversation between pa and $cinema1$, that are instances of the respective conversation protocols, after which the desired condition holds. Agent pa works on behalf of a user, thus it knows his credit card number ($\mathcal{B}^{pa} cc\_number$) and his desire to avoid using it in the transaction ($\neg\mathcal{B}^{pa} pay\_by(c\_card)$). It also believes to be an authority about the form of payment and about the user's credit card number and it believes that the other agents are authorities about what they communicate by inform acts. The initial mental state will also contain the fact that pa believes that no reservation for dinner nor for $nausicaa$ has been booked yet, $\mathcal{B}^{pa}\neg reservation(dinner)$ e $\mathcal{B}^{pa}\neg reservation(nausicaa)$, the fact that pa does not have a free ticket for the cinema yet, $\neg\mathcal{B}^{pa} cinema\_promo$, and some hypothesis on the interlocutor's mental state, e.g. the belief fluent $\mathcal{B}^{pa}\neg\mathcal{B}^{cinema1} cc\_number$, meaning that the web service does not already know the credit card number. In this context, the agent builds the following execution trace of *comp_services* ([[$restaurant$, R, $dinner$], [$cinema$, CIN, $nausicaa$]]):

$\mathsf{queryIf}(pa, restaurant1, available(dinner))$;
$\mathsf{inform}(restaurant1, pa, available(dinner))$;
$\mathsf{inform}(restaurant1, pa, reservation(dinner))$;
$\mathsf{inform}(restaurant1, pa, cinema\_promo)$;
$\mathsf{inform}(restaurant1, pa, ft\_number)$;
$\mathsf{queryIf}(pa, cinema1, available(nausicaa))$;
$\mathsf{inform}(cinema1, pa, available(nausicaa))$;
$\mathsf{queryIf}(cinema1, pa, cinema\_promo)$;
$\mathsf{inform}(pa, cinema1, cinema\_promo)$;
$\mathsf{inform}(pa, cinema1, ft\_number)$;
$\mathsf{inform}(cinema1, pa, reservation(nausicaa))$

We can easily see that there is no other execution trace of *comp_services* that satisfies the goals. This means that, in the search space defined by this procedure, it is not possible to use with success any other composition of the services (e.g. *restaurant1* with *cinema2*) and this does not depend on the possible outcomes of conversations. Observe that arbitrary compositions of the services, i.e. compositions that cannot be found as executions of *comp_services*, may satisfy the user's goal; in order to find them one could use a *general-purpose planner*. However, there are situations in which one has a *general schema* for the desired solution, which is helpful for reducing the search time.

3.8.3 Specifying and verifying systems of communicating agents in a temporal action logic

DyLOG is a sequential language which can describe the behavior of a single agent and prove *existential* properties, such as finding a sequence of actions achieving some goal. A more general problem is that of modelling systems of communicating agents, so as to be able to prove properties of the whole system. In this section we present a theory for reasoning about actions

which allows to describe the behavior of a network of sequential agents which coordinate their activities by performing common actions together[31, 32]. This theory is based on the Product Version of Dynamic Linear Time Temporal Logic (denoted $DLTL^{\otimes}$) [37], a logic which extends LTL, the propositional linear time temporal logic, by strengthening the *until* operator by indexing it with the regular programs of dynamic logic. Moreover, the formulas of the logic are decorated with the names of agents, thus allowing to describe the behavior of a network of agents which coordinate their activities by performing common actions together.

This logic provides a unified framework for specifying and verifying systems of communicating agents: Programs are expressed as regular expressions, (communicative) actions can be specified by means of action and precondition laws, properties of social facts can be specified by means of causal laws and constraints, and temporal properties can be expressed by means of the *until* operator.

Let us give a quick overview of the logic.

The logic $DLTL$ and its product version

First we recall the syntax and semantics of DLTL as introduced in [38]. $DLTL$ is an extension of LTL in which the next state modality is labelled by actions and the until operator is indexed by programs in Propositional Dynamic Logic (PDL) [36].

Let Σ be a finite non-empty alphabet whose members are interpreted as actions. Let $Prg(\Sigma)$ be the set of programs on Σ, defined as regular expressions. A set of finite words, representing computation sequences, is associated with each program by the mapping $[[]] : Prg(\Sigma) \rightarrow 2^{\Sigma^*}$.

Let $\mathcal{P} = \{p_1, p_2, \ldots\}$ be a countable set of atomic propositions. The set of formulas of DLTL(Σ) is defined as follows:

$$\text{DLTL}(\Sigma) ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha \mathcal{U}^{\pi} \beta$$

where $p \in \mathcal{P}$ and α, β range over DLTL(Σ), and π ranges over $Prg(\Sigma)$.

A model of DLTL(Σ) is a pair $M = (\sigma, V)$ where σ is an infinite sequence of actions and V is a valuation function. Given a model $M = (\sigma, V)$, a finite word $\tau \in prf(\sigma)$ (a finite prefix of σ), and a formula α, the satisfiability of a formula α at τ in M, written $M, \tau \models \alpha$, is defined as usual for the classical connectives. Moreover:

- $M, \tau \models \alpha \mathcal{U}^{\pi} \beta$ iff there exists $\tau' \in [[\pi]]$ such that $\tau\tau' \in prf(\sigma)$ and $M, \tau\tau' \models \beta$. Moreover, for every τ'' such that $\varepsilon \leq \tau'' < \tau'$, $M, \tau\tau'' \models \alpha$.

The formula $\alpha \mathcal{U}^{\pi} \beta$ is true at τ if "α until β" is true on a finite stretch of behaviour which is a computation sequence of the program π.

The derived modalities $\langle\pi\rangle$ and $[\pi]$ can be defined as follows: $\langle\pi\rangle\alpha \equiv \top\mathcal{U}^{\pi}\alpha$ and $[\pi]\alpha \equiv \neg\langle\pi\rangle\neg\alpha$. Furthermore $\bigcirc$ (next), $\Diamond$ and $\Box$ of LTL can be defined as follows: $\bigcirc\alpha \equiv \bigvee_{a\in\Sigma} \langle a\rangle\alpha$, $\Diamond\alpha \equiv \top\mathcal{U}^{\Sigma^*}\alpha$, $\Box\alpha \equiv \neg\Diamond\neg\alpha$.

Let us now recall the definition of $DLTL^{\otimes}$ from [37]. Let $Loc = \{1, \ldots, K\}$ be a set of *locations*, the names of the agents. A *distributed alphabet* $\tilde{\Sigma} = \{\Sigma_i\}_{i=1}^{K}$ is a family of (possibly non-disjoint) alphabets, where Σ_i is the set of actions which require the participation of agent i. If an action a belongs to Σ_i and to Σ_j, the two agents i and j will synchronize on this action. Let $\Sigma = \bigcup_{i=1}^{K} \Sigma_i$.

Atomic propositions are introduced in a local fashion, by introducing a non-empty set of atomic propositions $\mathcal{P}$. For each proposition $p \in \mathcal{P}$ and agent $i \in Loc$, p_i represents the "local" view of the proposition p at i, and is evaluated in the local state of agent i.

The formulas in $DLTL^{\otimes}(\tilde{\Sigma})$ are boolean combinations of formulas with the main constraint that no nesting of modalities $\mathcal{U}_i$ and $\mathcal{U}_j$ (for $i \neq j$) is allowed. A model of $DLTL^{\otimes}(\tilde{\Sigma})$ is a pair $M = (\sigma, V)$, where $\sigma \in \Sigma^{\infty}$ and $V = \{V_i\}_{i=1}^{K}$ is a family of functions V_i, where each V_i is the valuation function for agent i. The satisfiability of formulas in a model is defined as in DLTL, except that propositions are evaluated locally and the sequence of actions σ is projected on the alphabet of local actions of each agent.

Action theories and protocols

Given a set of communicating agents, each agent participating in an action execution has its own local description of the action determining the effects on its local state. The global state of the system can be regarded as a set of local states, one for each agent i. The *action laws* and *causal laws* of agent i describe how the local state of i changes when an action $a \in \Sigma_i$ is executed. The underlying model of communication is the synchronous one: the communication action $comm\_act(i, j, m)$ (message m is sent by agent i to agent j) is shared by agent i (the sender) and agent j (the receiver) and executed synchronously by them. Their local states are updated separately, according to their action specification. Though, for simplicity, we adopt the synchronous model, an asynchronous model can be easily obtained by explicitly modelling the communication channels among the agents as distinct locations.

A *protocol* defines the meaning of communicative actions involved in the conversation. In particular, by adopting a *social approach*, the protocol describes the effects of each action on the social state of the system. These effects, including the creation of new commitments, can be expressed by means of *action laws*. Moreover, the protocol establishes a set of preconditions on the executability of actions (*permissions*), which can be expressed by means of *precondition laws*. Each agent has a local view of the social state and the execution of a communicative action can in general affect both the state of the sender and the state of the receiver. In particular, all agents can see the effects on the social state of the actions to which they participate.

For instance, in the example of Section 3.8.1 there are two agents, the personal assistant *pa* and the web service *ws* providing ticket booking. The conversation protocol for the two agents will be given through a set of action laws and constraints in the form of permissions or commitments. Since our theory does not allow to express a global states, the protocol will be projected on the local states of the participating agents. Observe that, since the two agents participate in all communicative actions, they have the same local view of the social state, and of the action laws and constraints of the protocol.

Let us assume that *pa* is the sender of the following actions *queryIf(pa, ws, available(Film))*, *askBooking(pa, ws, Cinema)*, *give_cc(N)*, whereas the actions whose sender is ws are *inform(ws, pa, at(Film, Cinema))*, *inform(ws, pa, ¬ available(Film))*, *makeBooking(ws, pa, Cinema)*, *sendTicket(ws,pa)*. The effects of actions will be described by action laws such as (where $k = pa, ws$):

$$\Box_k([queryIf(pa, ws, available(Film))]_k asked(Film)$$
$$\Box_k([makeBooking(ws, pa, Cinema)]_k booked(Cinema)$$

where $asked(Film)$ and $booked(Cinema)$ are fluents of the social state.

Commitments can be effects of actions and will be represented by special fluents. They can be base-level commitments, of the form $C(ag_1, ag_2, action)$ (agent ag_1 is committed to agent ag_2 to execute the $action$), or they can be conditional commitments of the form

$CC(ag_1, ag_2, p, action)$ (agent ag_1 is committed to agent ag_2 to execute *action*, if the condition p is brought about).

For instance, when the web service finds a cinema, it commits to make the booking, if the customer asks it. Furthermore it commits to send a ticket if the customer gives its credit card number.

$$\Box_k([inform(ws, pa, at(Film, Cinema))]_k \\ CC(ws, pa, askedBooking(Cinema), makeBooking(ws, pa, Cinema)) \\ \wedge CC(ws, pa, cc\_given, sendTicket(ws, pa))$$

Some reasoning rules have to be defined for cancelling commitments when they have been fulfilled and for dealing with conditional commitments. For instance we can have the law (where $k = i, j$):

$$\Box_k((CC(i, j, p, a) \wedge \bigcirc_k p) \rightarrow \bigcirc_k(C(i, j, a) \wedge \neg CC(i, j, p, a)))$$

saying that a conditional commitment $CC(i, j, p, a)$ becomes a base-level commitment $C(i, j, a)$ when the condition p has been brought about. This law is a *causal law*.

The protocol can specify constraints (permissions) on the execution of actions by giving precondition to the actions. For instance *ws* will not send the ticket before the credit card number has been given:

$$\Box_k(\neg cc\_given \rightarrow [sendTicket(ws, pa)]_k \bot)$$

meaning that $sendTicket(ws, pa)$ cannot be executed in those states in which $\neg cc\_given$ holds, i.e. *cc_given* is a precondition of the action.

An agent i *satisfies its commitments* when, for all commitments $C(i, j, a)$ in which agent i is the debtor, the temporal formula:

$$\Box_i(C(i, j, a) \rightarrow \Diamond_i \langle a \rangle_i \top)$$

holds. Such a formula says that, when an agent is committed to execute action a, then it must eventually execute a.

Note that a protocol specified in this way is less rigid that the one given in Fig. 1, and can have different executions satisfying the action laws, preconditions and commitments. For instance the customer can leave the conversation before asking booking (the web service will have no base-level commitments to fulfill), or after asking booking and receiving confirmation, but before giving the credit card number (in this case the web service will only be committed to make booking, but not to send the ticket).

Reasoning about protocols

Given a protocol, we denote with $\mathcal{D}_i$ the *domain description* of agent i, i.e. its action laws and causal laws[3], with $Perm_i$ the set of precondition laws of the actions whose sender is i, and with Com_i the set of all temporal formulas, as the one above, describing the satisfaction of the commitments of agent i.

[3]Actually $\mathcal{D}_i$ must also model the frame problem. To deal with it we make use [31] of a completion construction which, given a domain description, introduces frame axioms for all the fluents in the style of the successor state axioms introduced by Reiter [58] in the context of the situation calculus.

If we do not know the behavior of any agent, we can only reason on the protocol by proving some properties of it, by assuming that all agents respect their permissions and commitments. This can be formalized as a validity check of the formula:

$$\bigwedge_j (\mathcal{D}_i \wedge Perm_j \wedge Com_j) \rightarrow p$$

where j ranges over all agents.

Assume instead that we know the behavior of some agents. For instance we are given a program (regular expression) π_{ws} which describes the behavior of the web service. In this case we would like to verify that *ws* always satisfies its social fact, i.e. its permissions and commitments. Since we don't know anything about the behavior of *pa*, we can only assume that it respects its social facts.

If $Prog_{ws}$ is the domain description of the behavior of ws, the following formula:

$$(\mathcal{D}_{ws} \wedge Prog_{ws} \wedge \mathcal{D}_{pa} \wedge Perm_{pa} \wedge Com_{pa}) \rightarrow (Perm_{ws} \wedge Com_{ps})$$

is valid if in all the executions of the system, in which agent ws respects its specification $Prog_{ws}$, and pa (whose internal program is unknown) respects the protocol specification (including its permissions and commitments), the permissions and commitment of agent ws are also satisfied. In general it is possible to prove that an agent is *compliant* (respects its "social facts") under the assumption that all other agents in the protocol are compliant.

Proofs and model checking

The above verification and satisfiability problems can be solved by extending the standard approach for verification and model-checking of Linear Time Temporal Logic, based on the use of Büchi automata. As described in [38], the satisfiability problem for DLTL can be solved in deterministic exponential time, as for LTL, by constructing for each formula $\alpha \in DLTL(\Sigma)$ a Büchi automaton $\mathcal{B}_\alpha$ such that the language of ω-words accepted by $\mathcal{B}_\alpha$ is non-empty if and only if α is satisfiable. Actually a stronger property holds, since there is a one to one correspondence between models of the formula and infinite words accepted by $\mathcal{B}_\alpha$. The size of the automaton can be exponential in the size of α, while emptiness can be detected in a time linear in the size of the automaton.

The validity of a formula α can be verified by constructing the Büchi automaton $\mathcal{B}_{\neg\alpha}$ for $\neg\alpha$: if the language accepted by $\mathcal{B}_{\neg\alpha}$ is empty, then α is valid, whereas any infinite word accepted by $\mathcal{B}_{\neg\alpha}$ provides a counterexample to the validity of α.

LTL is widely used to prove properties of (possibly concurrent) programs by means of *model checking* techniques. The property is represented as an LTL formula φ, whereas the program generates a Kripke structure (the model), which directly corresponds to a Büchi automaton where all the states are accepting, and which describes all possible computations of the program. The property can be proved as before by taking the product of the model and of the automaton derived from $\neg\varphi$, and by checking for emptiness of the accepted language.

Model checking can be adapted to the proof of the formulas given in the previous section, as follows [33]. Let us assume that the negation of a formula to be proved can be represented as $F \wedge \varphi$, where F contains the completion of the action and causal laws in the domain description and the initial state, and φ the rest of the formula. We can derive from F an automaton describing all possible computations, whose states are sets of fluents, which we consider as the model. In particular, we can obtain from the domain description a function $trans_a(S)$,

for each action a, for transforming a state in the next one, and then build this automaton by repeatedly applying these functions starting from the initial state. We can then proceed by taking the product of the model and of the automaton derived from φ, and by checking for emptiness of the accepted language. Note that, although this automaton has an exponential number of states, we can build it step by step by adopting a construction on-the-fly, similar to the construction used by model checkers based on LTL.

Bibliography

[1] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[2] 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.

[3] 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.

[4] 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*, Electronic Notes in Theoretical Computer Science. Elsevier Science Direct, 2004. to appear.

[5] M. Baldoni, C. Baroglio, and V. Patti. Web-based adaptive tutoring: an approach based on logic agents and reasoning about actions. *Artificial Intelligence Review*, 2004. To appear.

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

[7] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach. In *Proc. of ICTCS'2001*, volume 2202 of *LNCS*, pages 405–425. Springer, 2001.

[8] 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.

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

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

[11] C. Baral and T. C. Son. Approximate Reasoning about Actions in Presence of Sensing and Incomplete Information. In J. Małuszyński, editor, *Proc. of ILPS'97*, pages 387–404, Cambridge, 1997. MIT Press.

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

[13] BPEL4WS. http://www-106.ibm.com/developerworks/library/ws-bpel. 2003.

[14] 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.P. Müller, M. Wooldridge, and N.R. 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.

[15] J. Bryson, D. Martin, S. McIlraith, and L. A. Stein. Agent-based composite services in DAML-S: The behavior-oriented design of an intelligent semantic web, 2002.

[16] 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.

[17] R. Chinnici, M. Gudgin, J. J. Moreau, and S. Weerawarana. Web Services Description Language (WSDL) version 1.2, 2003. Working Draft.

[18] DAML-S. http://www.daml.org/services/daml-s/0.9/. version 0.9, 2003.

[19] 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.

[20] 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.

[21] 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.

[22] G. De Giacomo and R. Rosati. Minimal knowledge approach to reasoning about actions and sensing. In *Proc. of NRAC'99*, Stockholm, Sweden, August 1999.

[23] M. Denecker and D. De Schreye. Representing Incomplete Knowledge in Abduction Logic Programming. In *Proc. of ILPS '93*, Vancouver, 1993. The MIT Press.

[24] D.C. Dennett. *The Intentional Stance*. MIT Press, 1987.

[25] F. Dignum and M. Greaves. Issues in agent communication. In *Issues in Agent Communication*, volume 1916 of *LNCS*, pages 1–16. Springer, 2000.

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

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

[28] G. De Giacomo, Y. Lespèrance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

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

[30] L. Giordano, A. Martelli, and Camilla Schwind. Ramification and causality in a modal action logic. *Journal of Logic and Computation*, 10(5):625–662, 2000.

[31] L. Giordano, A. Martelli, and C. Schwind. Reasoning about actions in dynamic linear time temporal logic. In *FAPR'00, Int. Conf. on Pure and Applied Practical Reasoning*, Semptember 2000. Also in the Logic Journal of the IGPL, Vol. 9, No. 2, pp. 289-303, March 2001.

[32] L. Giordano, A. Martelli, and C. Schwind. Specifying and Verifying Systems of Communicating Agents in a Temporal Action Logic. In *Proc. of the 8th Conf. of AI*IA*, LNAI. Springer, 2003.

[33] L. Giordano, A. Martelli, and C. Schwind. Verifying communicating agents by model checking in a temporal action logic. In *Proc. of JELIA 2004*, LNAI. Springer-Verlag, 2004. to appear.

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

[35] F. Guerin. *Specifying Agent Communication Languages*. Phd thesis, Imperial College, London, April 2002.

[36] D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel, 1984.

[37] J.G. Henriksen and P.S. Thiagarajan. A product version of dynamic linear time temporal logic. In *Proc. of CONCUR'97*, 1997.

[38] J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied logic*, 96(1-3):187–207, 1999.

[39] A. Herzig and D. Longin. Beliefs dynamics in cooperative dialogues. In *Proc. of AMSTELOGUE 99*, 1999.

[40] F. Kabanza, M. Barbeau, and R.St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95:67–113, 1997.

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

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

[43] H. J. Levesque. What is planning in the presence of sensing? In *Proc. of the AAAI-96*, pages 1139–1146, 1996.

[44] 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.

[45] 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.

[46] 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.

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

[48] S. McIlraith and T. Son. Adapting Golog for Programming the Semantic Web. In *5th Int. Symp. on Logical Formalization of Commonsense Reasoning*, pages 195–202, 2001.

[49] S. McIlraith and T.C. 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.

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

[51] R. Moore. A formal theory of knowledge and action. In J. Hobbs and R. Moore, editors, *Formal theories of commonsense world*. Ablex, Noorwood, NJ, 1985.

[52] James H. Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering*, pages 121–140. Springer, 2001. http://www.fipa.org/docs/input/f-in-00077/.

[53] OWL. http://www.w3c.org/tr/owl-guide/. 2003.

[54] OWLS. http://www.daml.org/services/owl-s/. version 1.0, 2004.

[55] V. Patti. *Programming Rational Agents: a Modal Approach in a Logic Programming Setting*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 2002. Available at `http://www.di.unito.it/~patti/`.

[56] H. Prendinger and G. Schurz. Reasoning about action and change. a dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996.

[57] RDF. http://www.w3c.org/tr/1999/rec-rdf-syntax-19990222/. 1999.

[58] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[59] T.J. Rogers, R. Ross, and V.S. Subrahmanian. IMPACT: A system for building agent applications. *Journal of Intelligent Information Systems (JIIS)*, **14**, 2000.

[60] F. Sadri, F. Toni, and P. Torroni. Dialogues for Negotiation: Agent Varieties and Dialogue Sequences. In *Proc. of ATAL'01*, Seattle, WA, 2001.

[61] R. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proc. of the AAAI-93*, pages 689–695, Washington, DC, 1993.

[62] 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.

[63] S. Shapiro, Y. Lespérance, and H. J. Levesque. Specifying communicative multi-agent systems. In *Agents and Multi-Agent Systems - Formalisms, Methodologies, and Applications*, volume 1441 of *LNAI*, pages 1–14. Springer-Verlag, 1998.

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

[65] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. Heterogeneus Agent Systems: Theory and Implementation. MIT Press, 2000.

[66] M. Thielscher. Representing the Knowledge of a Robot. In *Proc. of the International Conference on Principles of Knowledge Representation and reasoning, KR'00*, pages 109–120. Morgan Kaufmann, 2000.

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

[68] WSDL. http://www.w3c.org/tr/2003/wd-wsdl12-20030303/. version 1.2, 2003.

Chapter 4

Business Rules

4.1 Introduction

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" [9]; they "formulate a law or custom that guides the behavior or actions of the actors connected to the organization" [6]. 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.

Before we continue to investigate the characteristics of business rules, we should clarify our interpretation of the term "organization" and how business rules and organizations are interconnected. A popular definition of the term *organization* is provided by Galbraith [34] who writes:

"First, organization emerges whenever there is a shared set of beliefs about a state of affairs to be achieved and that state of affairs requires the efforts of more than a few people. That is, the relationships among the people involved become patterned. The behavior patterns or structure derive from a division of labor among the people and a need to coordinate the divided work. Thus, a primary contribution of organization structure is to coordinate the interdependent subtasks which result from the division of labor. Second, the analysis and the points just made above introduce the essential attributes of organization and allow us to define what we mean by the term organization. We can say that organizations are (1) composed by people and groups of people (2) in order to achieve some shared purpose (3) through a division of labor (4) integrated by information-based decision processes (5) continuously through time".

This definition assumes that organizations are driven by shared goals and therefore emerge naturally in order to achieve those goals. However, what role do business rules play in this picture? Our answer is that business rules are the manifestations of the common (shared) will of the individuals who take part in an 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. Another important reason for explicit business rules is *efficiency*.

There are numerous examples that show the long tradition of formalized rules to govern organizations and business. Some are found in old Egypt and describe the construction of

pyramids, others describe the administration of the Chinese Empire of the Chou Dynasty, published 1100 B.C. [34]

Galbraith summarizes the main reasons in favor of using formalized rules as follows [34]:

- *Coordination*: In complex situations, the execution of tasks may need synchronization of the work done by several persons.
- *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.
- *Efficiency*: A machine(like) consistency of the task execution may lead to more efficient production, like in the automobile industry.
- *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, which are often described as graphical models, 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 has established a series of criteria which may help to decide which types of rules and tasks are eligible of formalization [65]:

- *Repetition rate*: A task is executed only once or may be repeated several times. If a certain repetition can be assumed, then the task is suited for being formalized as business rules. The repetition rate normally increases with the size of an organization, where it often decreases towards the upper levels of the organizations hierarchy.
- *Constancy*: A specific task can be executed in different ways depending on a specific situation. In a very unstable and dynamic environment, task execution is mostly less constant and therefore the task should not be formalized. However, if the constancy of repeated tasks is high, its formalization may be appropriate.
- *Complexity*: The complexity can be measured in regard to the number of elementary tasks to be interrelated. The higher the complexity of an organizations or a part of it is, the more need arises for coordination by e.g. formalization.
- *Determinability*: The execution of tasks can be more or less determined in advance. One extreme are non-deterministic tasks, whereas others are fully determined and may even be automated.

Rules that fit those criteria can be operationalized by transforming them into executable rule expressions, e.g. into a declarative logic based rule languages.

The remainder of this chapter is structured as follows: In the next section we will take a closer look into the various different types of formalizations of business rules, most notably

reaction rules, derivation rules and integrity constraints. In Section 4.3 we will survey the most prominent modern rule-based systems and their applications. In Section 4.4, we will give an overview of frameworks that aim to standardize and unify the notations of rules. The Section 4.5 will be devoted to the problem of conflicting rules and rule prioritizing. Finally, we discuss the possibilities and challenges for rule-based systems in today's web driven environment and we establish a list of use cases and requirements for the application of rules on the Semantic Web.

4.2 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 [70] and [72], which bases on [16] 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.

4.2.1 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 [72].

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.

In the following, we will discuss these three components of ECA rules in more detail.

The Event Component

ECA rules have been implemented for different application environments and for some time there was no common framework to describe the events signaled to the rule engines. However, along with some research projects in the field of active databases, implementation- and application-independent frameworks for the classification and detection of events have been proposed.

A prominent example is the event algebra Snoop [17], which is used as the event specification component of the Sentinel active database system [18]. Snoop is built around a concept of parameter (or event consumption) contexts, i.e. contexts that defined the semantics of events. This makes the approach extensible and widely applicable because new contexts with new event semantics can be introduced. Thus, applications of Snoop such as presented in [10], which applies Snoop to XML events, are possible.

In Snoop, an event is defined as an atomic occurrence, i.e. it happens completely or not at all. Further, events can be annotated by event modifiers like begin-of and end-of, which transform arbitrary time intervals into two logical instantaneous events.

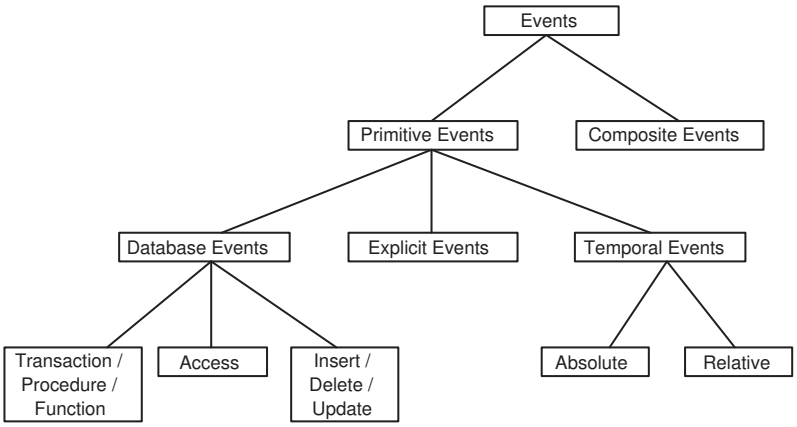


Figure 4.1: Snoop's event hierarchy [17]

A classification of the event types supported by Snoop is illustrated in Fig. 4.1. Snoop distinguishes primitive and composite events. Primitive events are finite sets of events that are pre-defined in the (application) domain of interest. Snoop foresees the following types of primitive events:

- *Database events*: These events represent the basic database management system (DBMS) operations, i.e. access, insertion, deletion and update of data. Further, the beginning and the end of database transactions, attached procedures and functions are also considered relevant events.
- *Explicit events*: These are events specific to the application environment and business logic of the system; they take place outside of the DBMS and need to be explicitly signaled to the system.
- *Temporal events*: These events are referring to points in time. They can be specified as an *absolute* time point (e.g. a cut-off date), or as a *relative* time point (e.g. the date of delivery of goods which depends on the date of the order).

Composite events are recursively constructed over other composite events and primitive events. The means for combining events to more complex events are Snoop's *event operators*, which include:

- *Disjunction* ($\vee$): The disjunction of two events E_1 and E_2, denoted as $E_1 \vee E_2$, occurs when E_1 occurs or E_2 occurs.
- *Sequence* (;): The sequence of two events E_1 and E_2, denoted as $E1; E2$, occurs when E_2 occurs provided E_1 has already occurred.
- *Conjunction*(Any, All): the conjunction event, denoted $Any(I, E_1, E_2, ..., E_n)$ where $I \leq n$, occurs when any I events out of the n events occur, ignoring the order of their occurrence. The *All* operator can be used to specify that $I = n$, i.e. that all the listed events have to occur.

- *Aperiodic operator* (A, A*): The event $A(E_1, E_2, E_3)$ occurs each time the event E_2 occurs within the temporal interval defined by the occurrences of E_1 and E_3. The event $A*(E_1, E_2, E_3)$ occurs only the *first* time this happens.

- *Periodic operator* (P, P*): The periodic operator is used to repeat an event during a specified interval $P(E_1, [t], E_3)$ where E_1 and E_3 are events and the constant t is the specification of the time between each event. Only those periodic events that occur within the interval between E_1 and E_3 are signaled.

Examples of the complex events that can modeled using these operators are shown below; we take the examples directly from [17] and use their simplified syntax to make the rules more readable:

Example 1: "Sample IBM stock every 30 minutes from 8 a.m. to 5 p.m. each day and compute the maximum value over a day". This rule can be written as follows:

```
On            P*(8 a.m., [30 mins.]: IBM-stock-price, 5 p.m.)
Condition     true
Action        compute maximum
```

Example 2: "When four withdrawals are made on an account in a day, do not allow further withdrawals". This rule can be expressed as follows:

```
On            A(8 a.m., Any(4, withdraw-on-an-account*), 5 p.m.)
Condition     true
Action        block further withdrawals
```

The Table 4.1 gives an overview over a variety of active database systems and compares the various properties of the event components of the systems to the Snoop/Sentinel approach. The tables shows which of the systems support database, temporal, explicit and complex events. Further, it shows which concepts support event modifiers and which support parameter contexts.

The Condition Component

If an event of an ECA rule is signaled to the rule engine, the condition of that rule will need to be evaluated in order to decide whether the rule is eligible to be fired or not. The separation of event and condition specifications provides both extensibility and a clear understanding of the "when" (an event), the "what" (a condition) and the "how" (an action).

A condition is a boolean function of data values, such as "the credit is greater than 100". A condition does not produce any side effect. It may be valid over an interval of time, while an event is atomic.

Conditions define *states* of a database or an expert system, and as such they can be interpreted as *guards* that protect transitions in a state-transition system. A guarded transition will only fire if its guard condition is fulfilled.

According to [43], the condition component can be classified into elementary and composite conditions:

- *Elementary conditions* are conditions on sets or literals (i.e. atomic predicates or negated predicates). An example of a set condition would be "if person X is element of the customer list". An example of a predicate condition is "if salary > 100".

Table 4.1: Overview of event concepts in the active database world (modified presentation of [17])

Feature / System	Primitive Event Types			Expressiveness		
	Database	Temporal	Explicit	Complex Events	Event Modifiers	Parameter Contexts
Ariel	Yes	Set of time values	No	No	Post	No
Interbase	Yes	No	No	Disjunction	Pre, Post	No
Postgres	Yes	No	No	Disjunction	equiv. to post	No
Starbust	Yes	No	No	Disjunction	Post	No
HiPAC	Yes	Absolute, relative	Yes	Disjunction, Disjunction, Sequence, Not	Pre, Post	No
Ode	Yes	Yes	Yes	Relative, Prior Sequence, Choose, Every, fa, faAbs, !	Before, After	No
ADAM	Yes	Timed events	No	No	Before, After	No
SAMOS	Yes	Explicite specification	Yes	Disjunction, Conjunction, Sequence, Not	Pre, Post	No
OSAM*	Yes	No	No	Disjunction	before, after	No
Snoop	Yes	Absolute, Relative	Yes	Or, Sequence, Any, Periodic, Aperiodic	User defined (begin-of, end-of, ...)	Recent, Chronicle, Continuous, Cumulative

- *Composite conditions* are constructed by applying boolean operators like conjunction and disjunction to the (elementary and composite) conditions to form more complex conditions.

The Action Component

The action component of an ECA rule specifies *what* the system is supposed to accomplish in case the specified event occurs and if the associated conditions holds. As in events and conditions, we can distinguish two kinds of actions [51]:

- an elementary action that is comprised of a single task to be carried out
- a composite action that is a structure – usually, a sequence – of several tasks to be executed.

Furthermore, we can distinguish actions by their use, i.e. by the effects they achieve [43]:

- *Data manipulation actions* encompass the creation, modification, retrieval, derivation and deletion of data. These data objects are not necessarily stored in a database but may also be stored outside the automated part of an information system (e.g. on a form). Insert and delete concern entire tuples and the other three may involve specific properties.
- *User actions* encompass a task which may be related to a data object but does not imply one of the operators mentioned above. User actions may contain any text, e.g. "call supplier and reorder product". A detailed syntax for their content (i.e. the specification of certain actions) is not always feasible.
- *Message actions* consist of a message to a processor of the information system. This message may be issued to an application or a human actor. A business rule whose action can be classified as a message action is called an *alerter*. The message can inform the target person on a situation and may trigger a specified action. Thus, the syntax for message actions contains at least a message and sometimes also the recipient of the message.

More on the definition of actions can be found in Chapter 3 of this report.

4.2.2 Production Rules

Production rules are similar to ECA rules; they may even be considered a special case of the general concept of reaction rules [72].

The term "production rules" was originally introduced in the context of formal grammars. For example, each rule in a Backus Naur form (BNF) specification of a context-free language is called a "production". However, here we use this term differently: 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. For example, the information "IF a person is walking and if the weather is rainy THEN this person needs an umbrella" can be represented by the following rule:

```
(defrule check-equipment
    (walking ?person)
    (rainy-weather)
  =>
    (assert (needs-umbrella ?person)))
```

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:

- *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).
- *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 [32]. 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 [54], which differs in several aspects of the organization of the internal working memory of the algorithms.

Examples for production rule systems are CLIPS and JESS, which will be described in more detail in Section 4.3.

4.2.3 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 [72].

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* [49], which is a well-known programming paradigm based on a subset of First Order Logic, named Horn clause Logic.

A program P is composed by a set of Horn clauses (i.e. implications) of the form $A \leftarrow B_1, ..., B_n$. Each such Horn clause can also be interpreted as a disjunction of literals with at most one positive literal, i.e. $\neg A \vee B_1 \vee ... \vee B_n$.

The execution of a program is driven by a query (or goal) of the form $\leftarrow B_1, ..., B_m$. Given a program P and a query $\leftarrow B_1, ..., B_m$, the purpose of an *execution* is to determine whether the conjunction $B_1 \wedge ... \wedge B_m$ is a logical consequence of the program P, i.e. whether $P \models B_1 \wedge ... \wedge B_m$

In the following, we briefly discuss programming environments that build on the idea of Logic Programming and that implement several different calculi for different types of logic programs, with slightly different semantics and application areas.

Prolog

Horn clauses form an important class of statements, since the consistency of a set of Horn clauses can be checked in a systematic and efficient manner. The most prominent example of

a language exploiting the advantageous properties of Horn clauses is Prolog. A first Prolog system has already been built in the early seventies and much of its underlying theory was subsequently provided by [47]. The calculus for reasoning over Prolog programs is called SLD; due to the nature of Horn clauses, SLD is both sound and complete, i.e. for all goal statements in this system that are really true, the calculus will prove that there is no refutation of it and that all statements the calculus determines to be true happen to be really true.

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. Logic programs that do not exploit such extra-logical constructs are often called Ordinary Logic Programs (OLP).

Datalog

A variant of Prolog, Datalog [20], 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 [25]. 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[25].

XSB Prolog

Another variant of Prolog is XSB, which is a research-oriented Logic Programming system for Unix and Windows-based platforms. In addition to providing all the functionality of Prolog, it contains several features not usually found in Logic Programming systems. Among these features are the availability of SLG resolution [21] and the handling of HiLog [22] terms.

SLG resolution is a bi-directional evaluation strategy, also called "tabling"; it uses partly top-down, partly bottom-up evaluation and ensures completeness for a large class of programs, enjoying the advantages of both evaluation schemas. The use of tabling allows for a different, more declarative programming style than Prolog that can be of use for a number of problems.

XSB also includes HiLog, a capability to process programs which have complex terms in predicates or functors. This allows programmers to program in a higher-order syntax, who can think of programming with parameterized predicates or with predicate variables.

These facilities significantly extend XSBs capabilities beyond those of a typical Prolog system, which justifies viewing XSB as a new paradigm for Logic Programming.

Smodels and DLV

Similar to XSB/SLG, which provides a more declarative style of programming than standard Prolog does, the Smodels [55] toolkit is based on an implementation-independent declarative semantics – the stable model semantics – which makes it much easier to develop applications because one does not have to worry too much about the internal implementation specific aspects of the system. Smodels is applied to range-restricted function free normal programs for which the stable model semantics is computable. Smodels can be used to generate stable models for such programs. In addition to model generation, Smodels can be used for query evaluation,

i.e. it is able to answer queries over the models generated, which makes it an alternative for traditional logic programming engines like Prolog.

Another inference engine for logic programs under the stable model semantics is DLV [23]. In contrast to Smodels, which is applied to normal programs, DLV is applied to disjunctive programs.

Both systems offer highly efficient implementations with attractive computational properties and they keep on being extended to handle priorities, cardinality constraints, weight constraints and numerous other features that are useful in many application domains.

Courteous Logics

Courteous Logics (CL) [39] extend Ordinary Logic Programs to include prioritized conflict handling, while maintaining computational tractability. Further, CL extends Ordinary Logic Programs with classical negation.

Each CL clause is an Ordinary Logic Program clause, i.e. it does not contain CUTs, etc. However, Courteous Logic Programs extend Ordinary Logic Programs by introducing a *labeling* of the clauses and by allowing users to use these clause labels to specify (meta-) rules that govern the conflict handling between the rules in the system.

Courteous Logics can be implemented on top of Logic Programming systems like Prolog; there exists a compiler that transforms CLPs to executable Prolog programs [39].

4.2.4 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 [70].

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".[70]
- *Structural assertions*: An important type of state constraints are structural assertions [42]. 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: $reserved \rightarrow allocated \rightarrow effective \rightarrow dropped{-}off$.[70]

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[70], c.f. also Section 4.3.1.

- structural assertions can be modeled as UML or entity/relationship diagrams and can be augmented by state constraints represented as OCL (Object Constraint Language) [74] expressions in the UML diagrams.

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.

4.3 Implementation of Rules in Information Systems

4.3.1 Rules in Active DBMS

Conventional (passive) database management systems (DBMS) solely serve as systems to store data in persistent data structures and to answer queries about the data stored. These passive DBMS do not actively perform any actions on their own.

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 have been defined as "database systems that respond automatically to events generated internal or external to the system itself without user intervention" [8].

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 [40], Postgres [68] and Starbust [75]. There do also exist object oriented active databases such as HiPac [26], Sentinel [18] and EXACT [28].

In most relational active DBMS, the event-, condition- and action-components of the ECA rules are implemented as follows:

- Events are the beginning or the end of SQL INSERT, UPDATE or DELETE operations. However, there exist implementations like [36] for Sybase or [48] for Sentinel that extend the scope of the DBMS by temporal and complex events.

- Conditions that determine whether the rule should be executed are represented as boolean SQL expressions. However, those conditions are to be evaluated not once but for all the tuples that may be affected by the operation. This means that some tuples may be changed by a trigger while others are left unchanged, because the condition did not apply to them.

- Actions to be taken are usually a sequence of SQL statements or whole database transactions; however, external programs and procedural attachments are also supported by many systems.

In the following we illustrate how to specify active rules as triggers in the commercial database system Oracle:

```
CREATE TRIGGER totalrevenue AFTER INSERT ON sales FOR EACH ROW
WHEN(NEW.id IS NOT NULL)
  UPDATE department
  SET revenue = revenue + NEW.amount
  WHERE NEW.dept = department.id
```

The trigger is named `totalrevenue` and the event it reacts to is an `INSERT` operation on the table `sales`. The trigger is called after the `INSERT` is performed; another option would be to call the trigger *before* or *instead* of the insert operation. The keyword `NEW` represents each of the inserted tuples. In the example it is used to access the value of field `amount` of the new sales entries, which is then added to a field `revenue` of the `department` which has generated the sale(s).

There are several options for how the triggered event is related to the evaluation of the rule's condition. There are three main possibilities for rule consideration [30]:

1. *Immediate consideration*: The condition is evaluated as part of the same transaction as the triggering event, and is evaluated immediately, either before, after or instead of executing the triggering event.

2. *Deferred consideration*: The condition is evaluated at the end of the transaction that included the triggering event. In this case, there could be many triggered rules waiting to have their conditions evaluated.

3. *Detached consideration*: The condition is evaluated as a separate transaction, spawned from the triggering transaction.

Similarly, there are several possibilities concerning the relationship between evaluating the rule condition and the execution of the rule action. The three possible options are again *immediate*, *deferred* and *detached* execution; most active systems use the first option, i.e. the action is immediately executed after the condition is successfully evaluated [30].

Besides the reactive behavior described above, modern database systems are able to capture and enforce another type of rules, i.e. integrity constraints, which have been laid out in Section 4.2.4. 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. Below we briefly illustrate those constraints:

- *Constraints on attributes*: Database systems allow to attach constraints to the fields definitions of tables. For instance, a modifier `not null` may be used to disallow NULL values for the defined attribute; the `unique` and `primary key` modifier force the field value for each tuple to be unique. Here an example of table definition that puts several constraints on the attributes of the table:

```
CREATE TABLE employee(
   id       INTEGER NOT NULL PRIMARY KEY,
   name     VARCHAR (5) NULL,
   projects SMALLINT NOT NULL DEFAULT 0
)
```

The `id` value has to be unique and must not be NULL, the `name` may be NULL and the `projects` value must not be NULL but has to be 0 (zero) as a default value.

- *Constraints on tables*: Some database systems allow to create constraints that go beyond the scope of single attribute constraints but rather may span over multiple attributes. For this purpose, the `CHECK` clause is used. In the example below, a constraint on the table `employee` is introduced which enforces that every employee is associated to precisely one `department` of the organization:

```
CREATE TABLE employee(
   id     NUMERIC(4) PRIMARY KEY,
   dept   VARCHAR (5)
      CHECK( dept IS NOT NULL AND
             1 = (SELECT COUNT(*)
                  FROM departments AS d
                  WHERE d.id = dept)),

)
```

- *Assertions on the data model*: to allow constraints with an even wider scope – spanning over the whole data model – the `CREATE ASSERTION` construct is provided by the SQL. The example below tells the DBMS to ensure the (overly simplified) policy that there must always be more projects than project managers in the organization:

```
CREATE ASSERTION haveProjects (
   CHECK ((SELECT COUNT(*) FROM projectmanger) <
          (SELECT COUNT(*) FROM projects))
)
```

- *Referential integrity constraints*: Another very popular type of constraint rules in databases are referential integrity constraints, which are enforced on so called "foreign keys", i.e. attributes whose values refer to keys of other (associated) tables. These constructs allow to define rules which tell the DBMS how to behave if a referenced value changes or gets deleted. In the example below, we tell the DBMS that the field `dept` of the table `employee` needs always be kept in sync with the corresponding value of the department's `id`:

```
CREATE TABLE EMPLOYEE(
   id     NUMERIC(4) PRIMARY KEY,
   dept   VARCHAR (5) NOT NULL,
   FOREIGN KEY (dept) REFERENCES departments(id)
     ON UPDATE CASCADE
)
```

In addition to the (integrity) constraints illustrated above, many relational DBMS support the creation of relational views, which can be seen as a (restricted) kind of derivation rule. For instance, to derive all accounting clerks from the organization's workforce, a view can be defined as follows:

```
CREATE VIEW accountingclerk (
  SELECT * FROM employee
  WHERE dept='acct')
```

4.3.2 Rule-Based Programming Environments

While the logic expressed in a rule can be written as imperative code, a rule-engine offers many benefits. Instead of locking the logic up in code written by developers, the logic can be moved out-board external to the actual application. In this way it is possible for non-developers to change the logic without having to rebuild the system. Additionally, by codifying all of the system rules in a central location, they are no longer scattered throughout the application. This allows for easier validation of the systems requirements and analysis of the logic of the system. In the following we briefly review popular rule engines.

Mandarax

Mandarax [29] 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).

Many rule engines used in production systems use data that is originally stored in (relational) databases. This requires considerable effort to keep the database and the rule-based systems in sync. Mandarax introduces a concept called "clause sets" to address this problem. Clause sets are basically iterators over collections of clauses. Such a clause set could be defined around an SQL query: the query returns a result set and the clause set builds facts from the records at query time. Replication of data is therefore not necessary any more.

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 [32]. 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 user manual discusses the advantages and disadvantages of their algorithm in comparison to Rete [29]; as a possible advantage over forward chaining systems, they argue that forward chaining systems are more difficult to integrate with databases, because all new or updated tuples would need to be propagated to the rule engine's memory-based fact base. Further, they argue that the performance advantage of forward chaining systems does not play out in most real world cases, since real world rule bases consist of a large number of facts compared to a much smaller number of rules – which is ideal for backward chaining rule systems.

Like Drools, Mandarax offers tight integration with the Java language. This allows for the reuse of business objects and logic and helps to increase the productivity of the software development process. Arbitrary Java methods can be regarded as *functions* in Mandarax, while

functions returning a boolean value can be considered as *predicates* in Mandarax. For instance, a Java expression `obj1.equals(obj2)` is interpreted as a logical fact `obj1 = obj2`.

While Mandarax does support RuleML as an input format, not all of its functionality is captured by RuleML. In particular, typing, complex terms and functions, clause sets and the integration of SQL data sources are not supported by RuleML.

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

An ILOG rule is composed of the following three parts: a header, a condition part and an action part. The header defines the name of the rule, its priority and packet name. The condition part (also called left hand side, LHS) of the rule defines the conditions that must be met such that the rule is eligible for execution. The action part is referred to as the right-hand side (RHS) of the rule and specifies the activities to be carried out when the rule is fired.

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.

ILOG offers support for the selection and handling of collections of objects which may be subject to rule processing. The advanced collection progressing allows objects to be accessed even if they do not directly reside in working memory. Instead, objects can be linked by other objects by fields or methods, and will be loaded dynamically by the system. This addresses one of the common disadvantages of forward chaining systems.

Jess

Jess [33] is a Java based rule engine and scripting environment inspired by the CLIPS [35] [62] expert system shell with its OPS5 [15] 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).

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

Jess can directly make use and manipulate Java objects. Moreover, it is a reference implementation for the JSR-94 standardization proposal [69], which aims at providing a uniform Java application programming interface to rule engines.

Like CLIPS, Jess is based on the Rete algorithm [32], 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 4.2.2).

Drools

Drools [24] is an implementation of the Rete algorithm tailored for the Java language, adapting it to an object-oriented interface that allows for a more natural expression of business rules with regards to business objects.

Drools is distributed under an Open Source license, and it offers an implementation of the JSR94 Standard API.

Drools offers an extensible set of different ways (semantic modules) to specify rules. These modules are centered around a unifying framework, the Drools Rule Language (DRL). The framework provides XML elements to structure the sets of rules, specify the input parameters of rules, their conditions and consequences and other properties.

The currently supported semantic modules built upon the DRL are Java, Groovy and Phyton. The Java module, for instance, allows for semantics based upon the Java programming language. Object types may be determined using Java classes while conditions and extractors are formulated in terms of Java expressions. The consequence of rules may be written as an arbitrary block of Java statements.

Drools offers several ways of conflict resolution. The simplest variant is the definition of an attribute "salience", where rules with higher salience values are given higher priority when ordered in the activation queue. In the event that multiple rules are assigned the same salience value, they are placed upon the queue in an arbitrary order. Another strategy is to initially order activations by their complexity as measured by the number of conditions in each rule. Rules with more conditions have a higher complexity and thus a higher priority when compared to rules with fewer conditions. When rules have the same complexity, ties are broken using their salience values. There does also exist the inverse form of this resolution concept, i.e. to chose rules with the lowest complexity first.

4.3.3 Rules in Imperative Program Code

While the declarative rule languages described in the paragraphs above are well suited for the governance of organizational rules, the largest amount of rules in today's information infrastructures is still stored in programs written in imperative and object oriented languages like COBOL (COmmon Business Oriented Language), various BASIC dialects, C, C++ and Java.

These languages allow the implementation of condition-action rules by the means of IF-THEN-ELSE statements or other selection constructs. Most of today's imperative languages also allow for the modularization of rules by organizing them into functions, procedures, modules and classes. The Java-based example below shows a function (method) that represents a rule determining the tax rate of a given customer:

```
public float getTaxRate(Customer c)
  switch(c.getCountry()) {
    case EU   : return 15.0;
```

```
    case USA  : return 10.0;
    case ASIA : return 12.0:
    default   : return 0.0;
  }
}
```

The use of languages like Java has many advantages; especially object oriented programming (OOP) languages are popular today, because OOP offers object polymorphism, method overloading, etc. which allow for the decomposition of complex structures into more manageable components.

On the other hand, the "hardcoding" of rules into program code has several disadvantages, one of them being the inability to change those rules without changing the source code.

Further, imperative and object oriented programs have a rigid execution flow, while rule engines apply relevant rules in much more flexible ways; often the standard sequential semantics of commands and methods does not allow for a simple, natural representation of the required control follow, which is supposed to be data- and inference-driven, as opposed to a rigid predefined order on rule application.

All these issues add heavy burden on developers (programmers), who need to find a way to capture the intended semantics of the business rules in imperative programs and then eventually, when the business requirements change, have to apply those changes in the program code and have to re-compile and re-deploy the application . Additionally, the transparency of the calculation is lost, which makes it difficult for the user to understand (and verify) the results of the program.

A first step to addressing these problems is usually to make the code *configurable*, i.e. to extract the relevant business rules from the program code and to store them in separate places where they can be changed during the runtime of the application. However, this actually requires writing a rule engine; but then it is unlikely that such an engine can be competitive with the highly optimized engines developed by the AI community in the last decades.

Therefore, the next logical step would be to employ dedicated rule engines to process the rules, which would reduce much of the necessary programming efforts, because all the rule handling algorithms would be outsourced to the rule engines with their highly optimized algorithms.

Especially rule engines like Jess, Drools or Mandarax are attractive for this job, because they can be tightly integrated with Java or other mainstream languages; this allows for reusing certain legacy code needed during the rule evaluation and execution. Similarly, there exist bridges between Prolog rule systems and Java (e.g. InterProlog [27] for XSB and SWI Prolog; Jasper for SICStus Prolog [66]) which allow to invoke Prolog from Java and to access Java objects from Prolog, and there exist tools like MINERVA [53], which provides Prolog's SLD calculus in Java, allowing for platform independent Prolog programs that can be tightly integrated with traditional Java programs.

This way, the advantages of both worlds – i.e. mainstream programming platforms and modern rule engines and logic programming environments – can be combined. Efforts like the upcoming JSR-94 standard [69] and numerous other rule APIs may help to achieve such hybrid architectures in a standardized way.

There already exist some tools for the extraction of business rules, e.g. the Cobol Transformation Toolkit (CORECT) [67].

4.3.4 Rules in End User Applications

As we discussed above in Section 4.3.3, it is a beneficial approach to make certain rules of an application accessible from outside of an application's source code to configure rules during runtime. This approach is followed by many applications, e.g. by e-mail applications like Microsoft Outlook and IBM Lotus Notes.

For instance, Lotus Notes allows to create rules for the handling of incoming e-mails. To create such an e-mail handling rule, the user has to specify:

- A set of conditions: For each condition, the user selects a property of incoming mails (e.g. sender, topic, priority) and a comparison operator (e.g. contains, does not contain, equals) and then enters the value to be compared with. Then the user adds the condition to the list of conditions of the mail handling rule.

- A set of actions: For each action, the user selects one of the available action items (e.g. move to folder, send carbon copy, delete) and for each of the actions selected, additional fields may need to be filled out (e.g. the carbon copy's recipient). Then the user adds the action to the list of actions of the mail handling rule.

After the rule is specified, the user may store the rule and switch it off or on. If the rule is switched on, Lotus Notes triggers the evaluation of the rule upon each incoming email, and if the condition is satisfied, the user-defined list of actions is executed automatically.

4.3.5 Rules in Business Process Descriptions

Business process descriptions lay out how agents (e.g. business partners, employees) interact in order to successfully accomplish a given goal. In this respect, business process descriptions fit our introductory definition of business rules (c.f. Section 4.1): they form statements about how a business is done and they govern the behavior of the involved participants, which can be both humans and information systems.

A business process description – often also called a workflow or business *protocol* – is comprised of a number of steps (tasks, activities), dependencies among those steps, routing rules, events and a description of the participants and their roles.

The formal description of (business) processes and other stateful systems has been subject of intensive research for many years. Well known formalisms to describe such dynamic systems are Petri Nets [60], Statecharts [41], UML sequence diagrams [58], the pi-Calculus [52] and various types of action logic (e.g. Concurrent Transaction Logic [14]).

These theoretical frameworks form the logical foundation of several of the currently relevant business process description languages, e.g. the Business Process Execution Language for Web Services (BPEL4WS) [71] (which is based on Petri Nets).

In the following we will describe how rules of the kind we outlined above in Section 4.2 can be found in the description of business processes. We use the classification presented in [59], who distinguish structure related, role related, message related, event related and constraint related rules.

Structure Related Rules

Structure related rules describe how the tasks of a process are interconnected with each other, i.e. how they are grouped and which interdependencies exist between the activities.

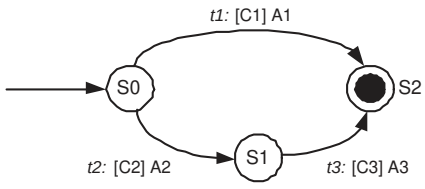


Figure 4.2: An automaton based abstraction of a business process

This kind of information is normally captured by the underlying process model, e.g. by a Petri Net or a Statecharts model. However, we can encode that information using rules as an alternative notation.

The transformation of an automaton to a set of rules (and facts) is a straightforward undertaking; for example, consider the automaton shown in Fig. 4.2; it is comprised of three states S0, S1, S2 (drawn as circles) and three transitions t1, t2, t3 (drawn as arrows), which describe how the process evolves. Each transition tn is guarded by a condition Cn and is associated to an action An, which will be executed during the transition. This semantics can be represented by the following set of facts and production rules:

```
(in-state S0)

(def-rule t1:              (def-rule t2:              (def-rule t3:
  (and (in-state S0)         (and (in-state S0)         (and (in-state S1)
       (C1))                      (C2))                      (C3))
   =>                         =>                         =>
  (execute-activity A1)      (execute-activity A2)      (execute-activity A3)
  (retract (in-state S0))    (retract (in-state S0))    (retract (in-state S1))
  (assert (in-state S2))     (assert (in-state S1))     (assert (in-state S2))
```

The fact `(in-state S0)` tells the rule engine that the described system is in state S0 initially. Each of the rules t1, t2, and t3 represent the transitions; for instance, rule t1 will fire if the system is in state S0 and if condition C1 is satisfied, just like the transition defined by the automaton. The execution of the rule t1 will lead to the transition from state S0 to state S2, along with the execution of the associated action A1.

Role Related Rules

Role related rules govern the participants that are involved in a process. In [59], *role assignment* rules are proposed to assign an activity to a certain role. For instance, in an e-commerce transaction, we could use a role assignment to assign the activity of the delivery service to a carrier.

```
if (delivery action is performed) then (Role-type is Carrier)
```

Other role related rule types are *role binding rules*, i.e. the assignment of a role to a particular business entity, and *event raiser rules*, which may trigger events related to the roles.

Message Related Rules

Message related rules regulate the use of information in a process; it is assumed that the entities propagate information by exchanging messages. A message is a data structure, for instance a tree based XML document. In [59], three types of message rules are proposed: *message distribution rules* which govern the distribution of messages , *message assignment rules* which assign messages to activities, and *message dependency rules* which can be used to derive the dependencies between messages.

As an example, consider the following rule:

```
if(FlightBookingActivity has Input)then (Message contains
(departureDate, arrivalDate, fromAirport, toAirport)
```

The rule above is a message assignment rule for the input of the flight booking activity, expressing that the input assigned to that activity has to contain departure and arrival date, and the departure and arrival airports.

Event Related Rules

Event rules govern the behavior of processes in reaction to expected or unexpected events. *Activity influence rules* regulate which activities are affected by which events. To determine which events are handled by which event handler activities (i.e. which activities should be carried out in case certain events occur), *event handler rules* can be defined.

As an example, consider the following rule:

```
if(delivery time exceeded) then (send email to customer)
```

This event handler rule indicates that if there are unforeseen troubles with the delivery, the customer must be notified.

Constraint Related Rules

These rules steer the use of constraints in a business process, represented by the pre- and post-conditions of the activities involved in the process. A pre-condition describes the requirements that must hold in order to achieve any of the activity's results. If a pre-condition is not fulfilled and an activity is still carried out, the results are undefined. A post-condition describes the effects the execution of the activities will achieve. If an activity is carried out with a valid pre-condition, we can expect that the effect formula will evaluate to true in the world state after the activity is completed, given that no errors occur.

The following rule is an example of a postcondition:

```
if(FlightBookinActivity is completed) then (Seat must be reserved)
```

A more thorough discussion of the specification of activities is given in Chapter 3.

4.4 Rule Language Frameworks

Most of the rule-based systems developed over the time have introduced their own concept of rules, along with proprietary notations to feed the rules into the systems. Those proprietary languages are well suited to reflect the capabilities, limitations and the intended use of their respective systems. On the other hand, a unifying framework to represent rules would be desirable, especially in a Semantic Web context, where rules are published on the Internet and agents may read and process those rules; a single unified markup syntax would ease the development and maintenance of web agents.

Besides language proposals like the ARML [19], which is an XML based language for the system-independent representation of reaction rules, there have also been efforts to provide *unified* syntactical concepts for many types of rules – not just of a single family or vendor. These approaches will be briefly discussed in the following paragraphs.

4.4.1 CommonRules

CommonRules [45] is a rule-based framework for developing rule-based applications with major emphasis on maximum separation of business logic and data, conflict handling, and interoperability of rules. Common rules was created as part of the "Business Rules for Electronic Commerce" project at IBM Research. The overall goal of CommonRules is to serve as a tool for the communication of executable business rules between enterprises using heterogeneous rule systems, and to enable incremental specification of executable business rules by non-programmers. Enterprises should be enabled to communicate their business policy rules about pricing, promotions, customer service provisions for refunds, ordering lead time, and other contractual terms and conditions, to a customer application, even when the seller's rules are implemented using a different rule system than the buyer's system.

CommonRules provides a common "interlingua" rule representation for the exchange of rules between heterogeneous rule representations employed in various rule-based applications. It uses an XML based interchange format for rules, called Business Rules Markup Language (BRML) [57], that corresponds to this interlingua and that can be seen as a predecessor of RuleML. BRML's expressive class is situated courteous logic programs, i.e., declarative logic programs with negation-as-failure, (limited) classical negation, prioritized conflict handling, and disciplined procedural attachments for queries and actions. BRML's semantics is based on Logic Programs and captures a common core shared by many commercially important rule systems, including relational database systems, logic programming systems, production rule systems and event-condition-action rule systems.

CommonRules includes sample translators between the BRML XML interchange format and several existing rule systems. Developers (i.e. rule system vendors) can write their own such translators. CommonRules also includes a Courteous Compiler (c.f. Section 4.2.3) that implements the Courteous enhancement via a pre-processor that can be added modularly to a variety of existing commercial rule systems.

4.4.2 Rule Markup Language (RuleML)

RuleML [13] [63] 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

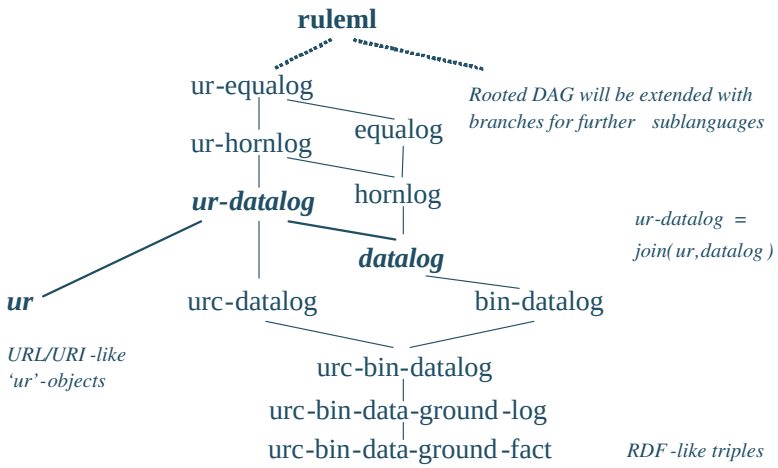


Figure 4.3: The RuleML hierarchy with 12 derivation-rule sublanguages [13]

(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. The Fig. 4.3 shows the various variants of derivation rules known in the literature. Based on that ontology, concrete syntaxes have been crafted to express derivation rules in 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) [44], 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 4.3). 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) [12]. Recent efforts also went into defining MOF-RuleML [73], 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 [72]. 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].

4.5 Dealing with Rule Conflicts and Inconsistency

4.5.1 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 two different ways:

1. 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.

2. 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 [50] [3].

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.

4.5.2 What is A Conflict?

In the simpler case, a conflict is directly represented by logical negation: two rules are conflicting if the head of one rule is the negation of the other.

However, there are more general cases which arise often in practice. For example, an investment consultant may base her recommendation on three levels of risk investors are willing to take: low, moderate and high. Obviously, only one level of risk per investor is allowed to hold at any given time. Thus, a rule suggesting reasons why low risk is appropriate for a particular investor is conflicting with any rule with head medium or high risk.

In general, sets of atoms may be declared to be mutually exclusive.

4.5.3 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.

The next dichotomy is between credulous and sceptical approaches (terms used extensively in the area of Nonmonotonic Reasoning). According to the first idea, conflicting rules should fire and contradictory conclusions should be drawn, albeit without a trivialization effect: we should not be able to conclude everything even if we have derived conclusions A and its negation. The interpretation of such conclusions is that they are at least supported: there is reasoning that supports the conclusions, though there may also be reasoning that concludes its negation.

On the other hand, sceptical approaches accept as conclusions only indisputable facts: a rule can only be applied if possibly existing conflicting rules are inapplicable. Such approaches are useful in situations where the cost of drawing a false decisions is higher than the cost of not drawing conclusions.

Reasoning systems falling in this category are wide-spread in logic programming, knowledge representation and the Semantic Web [2] [4] [5] [61].

4.5.4 Resolving Conflicts Using Priorities

If a sceptical view is taken, the case of rules not being applied is quite common. One way of resolving such 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] [5] [56] [64]. 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 [38] [7].

4.5.5 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 [11].

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.

4.6 Business Rules for the Semantic Web

In Section 4.3.3 we have discussed the advantages of explicit formal rules and dedicated rule engines over hard-coded rules in programs: formal rules offer more flexibility, can be more easily adapted during runtime and since they are easier to be read and analyzed.

While rule-based systems are very common in the traditional software markets, such as production planning systems, enterprise information systems or end user software such as described in Section 4.3.4, another new form of application delivery is currently developing: the Web, along with emerging technologies like Web services and the Semantic Web. Software built on top of those new environments are often characterized as being based on *loosely coupled* components, in contrast to traditional software systems, which are commonly delivered as monolithic blocks of software.

Furthermore, on the Web there is a tendency for very *heterogenous* user groups and heterogenous hard- and software is used. In addition, the data sources of the web are not centralized, leading to heterogeneity and volatility of the data (frequent changes, differing data schemas, differing levels of service quality, etc.) to be processed; this poses another challenge to Web based computing, for instance in the context of bio-informatics (cf. Working Group A-2 of this project).

All those issues described above demand additional flexibility of the software, which, when produces by traditional means, would place a huge burden on the developers, maintainers and users of the software. Therefore, we suggest the application of rule-based techniques to build Internet based systems. The rationale behind this is that declarative rules provide a higher degree of flexibility and adaptivity of the applications, which, as pointed out above, is essential for Internet based applications.

4.6.1 Business rules and Web Agents

Business rules in the context of the Web can be seen as declarative descriptions that steer the behavior of (semantic) Web agents.

As Fig. 4.4 illustrates, a (semantic) Web agent may receive input from Web data sources (e.g. an XML based data feed the agent is subscribed to), from other agents (e.g. SOAP Web services it uses for certain tasks), from the human user (who might provide input via some user interface including voice recognition) and, there might even be input generated by the rule-based system itself (e.g. a production rule triggered by a timer event). In short, the behavior of Web agents can be largely defined by rules. A standard notation for such rules would be desirable to allow the exchange and analysis of such rules.

4.6.2 The Need For a Web Rule Language

However, as discussed in Section 4.4, no such universally usable rule language for the (Semantic) Web exists yet. Therefore, it is a goal for future research to establish a model of the most relevant rule types (i.e. derivation rules, reaction rules, production rules), which should fulfil the following properties:

- there should exist a usable and intuitive XML based serialization format
- the language should be independent of the run time environments where the rules may be executed

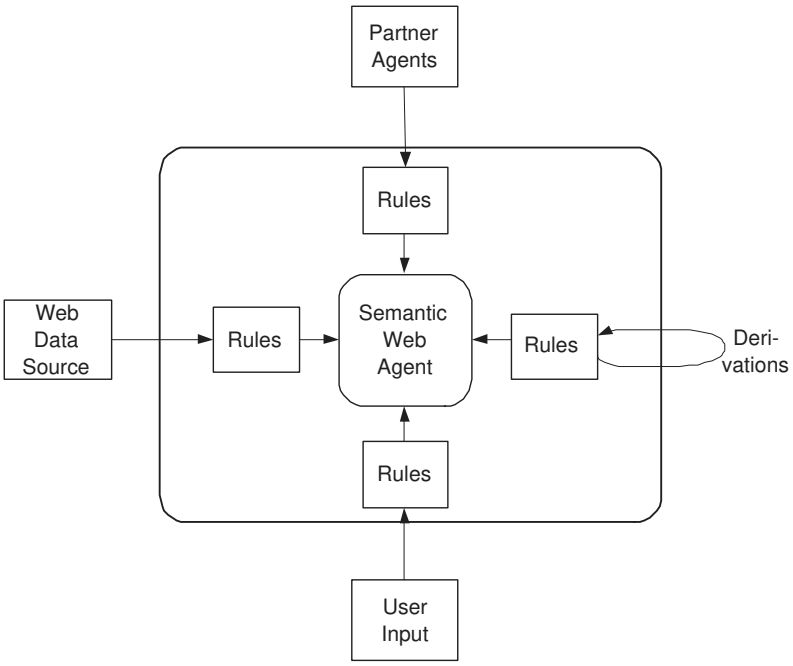


Figure 4.4: A rule-based Web agent

- the conceptual model of the language should be formally and precisely defined, such that semantics-preserving translations (e.g. to proprietary rule languages supported by certain tools) are possible.
- the rule language should be ontology aware, i.e. there should be a notion of concepts (classes) and roles (properties) to structure the data the rule engine operates upon.

4.6.3 An Example

A representative use case for rule-based Web agents was described by Gerd Wagner in [1]. This use case describes a personal portfolio software agent, which monitors the price development of the shares of the portfolio of its owner and reacts in response to significant drops in value, e.g. by sending an alert to its owner.

The following predicates are used: "is exempt from profit taxes" (DR1) and "is significant" (DR2). They are defined by means of the following two derivation rules:

1. DR1: An investment is exempt from profit taxes, if it is for more than 1 year in the portfolio.
2. DR2: An investment is significant, if the value of the investment in the portfolio is more than 10

The behavior of the agent is specified by the following five reaction rules:

- Share drop and no profit taxes: If a share price dropped by more than 5% and the corresponding investment is exempt from profit taxes, then sell the investment.
- Share drop with profit taxes: If a share price dropped by more than 5% and the corresponding investment is not exempt from profit taxes, then send an alert with high priority.
- Share drop, significant investment with no profit taxes: If a share price dropped by more than 3% and the corresponding investment is significant and is exempt from profit taxes, then sell the investment.
- Share drop, significant investment with profit taxes: If a share price dropped by more than 3% and the corresponding investment is significant and is not exempt from profit taxes, then send an alert with high priority.
- Share drop, non significant investment: If a share price dropped by more than 3% and the corresponding investment is not significant, then send an alert.
- Share crosses predicted boundaries: If a share dropped under a certain predefined price, or its price has gone over a certain predefined value, notify brokers.

Bibliography

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

[2] Alferes, J., Pereira, L.M.: Reasoning with logic programming. In: LNAI 1111, Springer-Verlag (1996)

[3] Antoniou, G.: Nonmonotonic Reasoning. The MIT Press (1997)

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

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

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

[7] Bassiliades, N., Antoniou, G., Vlahavas, I.: 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)

[8] Bauzer-Medeiros, C., Pfeffer, P.: A mechanism for managing rule in an object-oriented database (1991)

[9] Bell, J., Brooks, D., Goldbloom, E., Sarro, R., Wood, J.: Re-Engineering Case Study Recommendsations to Application Developers. US West Information Technologies Group. Bellevue Golden (1990)

[10] Bernauer, M., Kappel, G., Kramler, G.: Composite events for xml. In: Proceedings of the International WWW Conference, New York, USA. (2004)

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

[12] Boley, H., Grosof, B., Sintek, M., aid Tabet, Wagner, G.: Object-Oriented RuleML, version 0.85 of 15 march 2004, http://www.ruleml.org/indoo (2004)

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

[14] Bonner, A.J., Kifer, M.: Concurrency and communication in transaction logic. In: Joint International Conference and Symposium on Logic Programming, 142–156 (1996)

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

[16] Bubenko, J.A., Brash, D., Stirna, J.: Ekd - enterprise knowledge development user guide (1998)

[17] Chakravarthy, S., Mishra, D.: Snoop: An expressive event specification lanauge for active databases (1993)

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

[19] Cho, E., Park, I., Hyum, S.J., Kim, M.: ARML: an active rule mark-up language for heterogeneous active information systems. In: First International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML-2002). (2002)

[20] Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Surveys in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (1990)

[21] Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM, 1:20–74 (1996)

[22] Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. Journal of Logic Programming, 3:187–230 (1993)

[23] Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The dlv system: Model generator and application frontends. In Bry, F., Freitag, B., Seipel, D., eds.: Proceedings of the 12th Workshop on Logic Programming (WLP'97). Research Report PMS-FB10, 128–137 (1997)

[24] Codehaus: Drools: Object-Oriented Rule Engine for Java, http://drools.org/ (2004)

[25] Date, C.: An Introduction to Database Systems. Addison-Wesley (1995)

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

[27] Declarativa: InterProlog product site, http://www.declarativa.com/interprolog/ (2004)

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

[29] Dietrich, J.: The mandarax manual, http://mandarax.sourceforge.net /docs/mandarax.pdf (2003)

[30] Elmasri, R., Navathe, S.: Fundamentals of Database Systems, 3rd edition. World Student Series. Addison-Wesley (2000)

[31] Eshuis, R.: Semantics and Verification of UML Activity Diagrams for Workflow Modelling, PhD thesis, University of Twente (2002)

[32] Forgy, C.L.: Rete: A fast algorithm for the many pattern / many object pattern match problem. Artificial Intelligence, 1:17–37 (1982)

[33] Friedman-Hill, E.: Jess in Action. Manning Publications Co. (2003)

[34] Galbraith, J.: Organization Design. Addison-Wesley (1997)

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

[36] Gopalakrishnan, G.A.: Making sybase fully active: supporting composite events and prioritized rules (2002)

[37] Grosof, B.: Prioritized conflict handling for logic programs. In: Proc. International Symposium on Logic Programming (ILPS-97). (1997)

[38] Grosof, B., Gandhe, M., Finin, T.: Sweetjess: Translating damlruleml to jess. In: Proc. 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)

[39] Grosof, B.N., Labrou, Y., Chan, H.Y.: A declarative approach to business rules in contracts: Courteous logic programs in xml. In Wellman, M.P., ed.: Proceedings of the 1st ACM Conference on Electronic Commerce, EC-99, Denver, CO, USA, ACM Press (1999)

[40] Hanson, E.: Rule condition testing and action execution in ariel. (1992)

[41] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming (1987)

[42] Hay, D., Healy, K.A.: Defining business rules – what are they really? (2000)

[43] Herbst, H.: Business Rule-Oriented Conceptual Modeling. Contributions to Management Science. Physica/Springer Verlag (1996)

[44] Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web rule language combining OWL and RuleML, version 0.5 of 19 november 2003, http://www.daml.org/2003/11/swrl/ (2003)

[45] IBM: Commonrules at alphaworks, http://www.alphaworks.ibm.com/ tech/commonrules (2002)

[46] ILOG Inc.: ILOG web site, http://www.ilog.com (2004)

[47] Kowalski, R.A.: Predicate logic as programming language. In Rosenfeld, J.L., ed.: Proceedings of IFIP Congress, North-Holland Publishing Company (1974)

[48] Lee, H.: Support for temporal events in sentinel: Design, implementation, and preprocessing (1996)

[49] Lloyd, J.W.: Logic Programming. Springer Verlag (1984)

[50] Marek, V., Truszczynski, M.: Nonmonotonic Reasoning. Springer Verlag (1993)

[51] Mertens, P., Hofmann, J.: Aktionsorientierte datenverarbeitung. Informatik-Spektrum, 9:323 – 333 (1986)

[52] Milner, R.: Communication and Concurrency. Prentice Hall (1989)

[53] IF Computer: MINERVA product site, http://www.ifcomputer.com/minerva/ (2004)

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

[55] Niemelae, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning. volume 1265 of Lecture Notes in Artificial Intelligence, Springer Verlag, 420–429 (1997)

[56] Nute, D.: Defeasible logic. (In: Handbook of Logic for Artificial Intelligence and Logic Programming, Vol. III)

[57] Oasis: Business rules markup language (BRML), oasis cover pages, http://xml.coverpages.org/brml.html (2002)

[58] OMG UML Task Force: Unified Modeling Language (tm), http://www.omg.org/cgi-bin/doc?ad/2002-06-18 (2002)

[59] Orriens, B., Yang, J., Papazoglou, M.: A framework for business rule driven service composition. In: 4th VLDB Workshop on Technologies for E-Services. (2004)

[60] Petri, C.A.: Kommunikation mit Automaten. Dissertation (1962)

[61] Reiter, R.: A logic for default reasoning. Artificial Intelligence, 13:81–132, (1980)

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

[63] RuleML Initiative: The Rule Markup Initiative Web Site, http://ruleml.org/ (2004)

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

[65] Schmidt, J.W.: Planvolle Steuerung gesellschaftlichen Handelns. Verlag fr Sozialwissenschaften (1983)

[66] SICS: SICStus product site, http://www.sics.se/sicstus/ (2004)

[67] SoftwareMining: vendor website, http://www.softwaremining.com (2004)

[68] Stonebraker, M., Hanson, E., Hong, C.H.: The design of the postgres rule system. In: 3rd International IEEE Conference on Data Science. (1987)

[69] Sun Microsystems: (Java specification request jsr-94 - java rule engine api, http://www.jcp.org/en/jsr/detail?id=94)

[70] Taveter, K., Wagner, G.: Agent-oriented enterprise modeling based on business rules. In: Proc. of 20th Int. Conf. on Conceptual Modeling (ER2001). LNCS, Springer-Verlag (2001)

[71] Thatte, S.: Business process execution language for web services version 1.1, 05 May 2003 (2003)

[72] Wagner, G.: How to design a general rule markup language? In: XML Technologien für das Semantic Web - XSW 2002, Proceedings zum Workshop, 24.-25 Juni 2002, Berlin. (2002)

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

[74] Warmer, J., Kleppe, A.: The Object Constraint Language : Precise Modeling with UML. Addison-Wesley Pub Co (1998)

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

Chapter 5

Controlled Natural Languages

5.1 Introduction

The following two quotations perfectly express our motivation to use a controlled language within REWERSE.

"[...] a truly semantic web is more likely to be based on natural language processing than on annotations in any artificial language." – John F. Sowa, CG Mailing List, October 19, 2003.

"Controlled Natural Languages are subsets of natural languages whose grammars and dictionaries have been restricted in order to reduce or eliminate both ambiguity and complexity. Traditionally, controlled languages fall into two major categories: those that improve readability for human readers, particularly non-native speakers, and those that improve computational processing of the text." [1]

5.2 Attempto Controlled English

5.2.1 What is Attempto Controlled English?

Attempto Controlled English (ACE) is a specification and knowledge representation language[1]. ACE is a subset of English, meaning that all ACE sentences are correct English, but that not all English sentences are allowed in ACE. ACE texts are computer-processable and can be unambiguously translated into full first-order logic. ACE appears perfectly natural, but is in fact a formal language with the semantics of the underlying first-order language. In brief, ACE combines the familiarity of natural language with the rigour of formal languages.

Attempto Controlled English and the Attempto system are intended for domain specialists - e.g. engineers, economists, physicians - who want to use formal methods, but may not be familiar with them. Thus the Attempto system has been designed in a way that allows users to work solely on the level of ACE without having to take recourse to its internal logic representation.

The use of ACE presupposes only basic knowledge of English grammar. However, being a formal language ACE must be learned, and as experience has shown it can be learned in a short time.

[1] cf. http://www.ifi.unizh.ch/attempto

5.2.2 Attempto Controlled English in a Nutshell

The following is intended as a quick overview of the main features of the language ACE. The complete language is described in the ACE Language Manual[2].

Vocabulary

The vocabulary of ACE comprises

- function words (e.g. determiners, conjunctions, prepositions), and
- content words (nouns, verbs, adjectives, adverbs).

While function words are predefined by the Attempto system, users can define new content words or modify existing ones with the help of a lexical editor. Alternatively, users can import existing lexica. As a result, the Attempto vocabulary can be custom-tailored to the needs of the respective domain. Furthermore, users can define aliases for a content word and attach comments that explain the meaning and use of a word.

Grammar

The grammar of ACE defines and constrains the form and the meaning of ACE sentences via construction and interpretation rules.

An ACE text consists of a sequence of sentences. There are

- simple sentences, and
- composite sentences.

Furthermore, there are queries that allow users to interrogate the contents of ACE texts.

Simple Sentences Simple sentences are built according to the following construction rule

subject + *verb* + *complements* + *adjuncts*

Complements (objects) are necessary for transitive and ditransitive verbs, whereas adjuncts (adverbs or prepositional phrases) are optional. Here is a simple sentence.

A customer inserts a card.

All elements of a simple sentence can be elaborated upon to describe a given situation in greater detail. To further specify the nouns *customer* and *card* we could add adjectives

A trusted customer inserts a valid card.

possessive nouns and of-prepositional phrases

John's customer inserts the card of Mary.

or variables and quoted strings as appositions

[2]cf. http://www.ifi.unizh.ch/attempto

The customer X gets a message "Invalid card.".

Other modifications of nouns are possible through relative sentences that are described below. We can also detail the verb, e.g. by adding an adverb

A customer inserts a card manually.

or by adding prepositional phrases, e.g.

A customer inserts a card into a slot.

We can combine the above enhancements to arrive at the sentence

John's customer who is new manually inserts a valid card of Mary into a slot A.

that in spite of its complexity is still a simple sentence.

Composite Sentences

Composite sentences are recursively built from simpler sentences through coordination, subordination, quantification, and negation. Coordination by *and* and *or* is possible between sentences and between phrases of the same syntactic type

A customer inserts a card and the machine checks the code. A known and trusted customer enters a card and a code.

Coordination by *and* and *or* is governed by the standard binding order of logic, i.e. *and* binds stronger than *or*. The sentence

A customer inserts a VisaCard or inserts a MasterCard and types a code.

means that the customer inserts a VisaCard, or the customer inserts a MasterCard and types a code. Commas can be used to override the standard binding order. Consequently, the sentence

A customer inserts a VisaCard or inserts a MasterCard, and types a code.

means that the customer inserts a VisaCard and types a code, or inserts a MasterCard and types a code.

There are two forms of subordination: relative sentences and if-then sentences. Relative sentences starting with *who, which, that* allow to add detail to nouns, e.g.

A customer who is new inserts a card that he owns.

With the help of if-then sentences we can specify conditional or hypothetical situations, e.g.

If a card is valid then a customer inserts it.

Quantification allows us to speak about all objects of a class, or to denote explicitly the existence of at least one object of a class. To express that all customers insert cards we can write

Every customer inserts a card.

or alternatively

Each of the customers inserts a card.

Both sentence mean that each customer inserts a card that may, or may not, be the same as the one inserted by another customer. To specify that all customers insert the same card - however unrealistic that situation may seem - we can write

There is a card that every customer inserts.

ACE does not know the passive voice. To state that every card is inserted by a customer we can write in a somewhat stilted way

For each of the cards there is a customer who inserts it.

The textual occurrence of a quantifier opens its scope that extends to the end of the sentence, or – in coordinations – to the end of the respective coordinated sentence. In the case of several quantifiers this rule leads to the correct nesting of the scopes of the quantifiers.

In addition to these and other existential and universal quantifiers, ACE also provides various constructs for plurals – for instance *the cards, two cards, a card and a code, two kilos of apples* – and generalised quantifiers - such as *at least, at most, less than, more than.*

Negation allows us to express that something is not the case, e.g.

A customer does not insert a card.
A card is not valid.

To negate something for all objects of a certain class one uses *no*

No customer inserts a card.

or, equivalently, *there is no*

There is no customer who inserts a card.

ACE provides further forms of negation, for instance *it is not the case that, not every, not all, nobody* etc.

Query Sentences

Query sentences permit us to interrogate the contents of an ACE text. Query sentences come as yes/no-queries and as wh-queries.

Yes/no-queries establish the existence or non-existence of a specified situation, for instance

Does a customer insert a card?

With the help of wh-queries, i.e. queries with query words, we can interrogate a text for details of the situation described. If we specified

A new customer inserts a valid card manually into the slot at 9 o'clock.

we can ask for each element of the sentence, e.g.

Who inserts a card?
Which customer inserts a card?
What does the customer insert?
How does the customer insert a card?
When does the customer insert a valid card?

Note, however, that we cannot ask for the verb itself.

5.2.3 Constraining Ambiguity

To constrain the ubiquitous ambiguity of full natural language ACE employs three simple means

- some ambiguous constructs are not part of the language; unambiguous alternatives are available in their place,
- all remaining ambiguous constructs are interpreted deterministically on the basis of a small number of interpretation rules; the interpretations are reflected in a paraphrase,
- users can either accept the assigned interpretation, or they must rephrase the input to obtain another one.

Note that ACE only handles structural ambiguity, but not lexical ambiguity. Altogether there are about a dozen interpretation rules in ACE, one of which is the interpretation rule for quantifiers we already encountered above. Here is an interpretation rule pertaining to relative sentences. In full natural language relative sentences combined with coordinations can introduce ambiguity, e.g. given the sentence

A customer inserts a card that is valid and opens an account.

it is not immediately clear whether the customer or the card opens the account. In ACE, however, the sentence has the unequivocal meaning that the customer opens the account. This is reflected in the paraphrase by curly brackets

A customer inserts {a card that is valid} and opens an account.

To express the alternative – though not very realistic – meaning that the card opens the account the relative pronoun that must be repeated, thus yielding

A customer inserts a card that is valid and that opens an account.

with the paraphrase

A customer inserts {a card that is valid and that opens an account}.

Users who are learning ACE will soon realise that a sentence that in full English would be ambiguous is unambiguous in ACE.

5.2.4 Anaphoric References

An ACE text consist of a sequence of sentences interrelated by anaphoric references, e.g.

John is a customer. He inserts a card that belongs to himself and types a code X. Bill sees it. He inserts his own card and types X. The code X is invalid.

In ACE anaphoric references – via pronouns, variables and definite noun phrases – can only refer to preceding noun phrases. Proper names always refer to the same person or object.

During the processing of the text the ACE parser replaces each anaphoric reference by the most recent accessible noun phrase that agrees in gender and number, and displays the replacements in a paraphrase

John is a customer. [John] inserts a card that belongs to [John] and types a code X. Bill sees [the code X]. [Bill] inserts [Bill's] card and types [the code X]. [The code X] is invalid.

The search for antecedents of anaphora is governed by accessibility restrictions. For instance, in the sentences

John does not own a card. He enters it.

it cannot refer to *a card*. Neither in

Every customer owns a card. It is correct.

In both cases the noun phrase *a card* is not accessible from the outside of the sentence in which it occurs.

5.2.5 Domain Knowledge

The Attempto system is not associated with any specific application domain, nor with any particular formal method. By itself it does not contain any knowledge or ontology of the intended domain, of formal methods, or of the world in general. Thus users must explicitly define domain knowledge through ACE sentences like

A card is valid.

In this sentence the words *card* and *valid* are processed by the Attempto system as uninterpreted syntactic elements, i.e. any real world interpretation of these words is solely performed by the human writer or reader. Whatever understanding we may have of the concepts *card* and *valid* is not part of the ACE sentence, unless we decide to add information explicitly, e.g. by sentences like

Every card that carries a code is valid.

In summary, the only source of information about an ACE text is the text itself. However, see below how the Attempto Reasoner RACE uses additional knowledge about the English language and about natural numbers in the form of auxiliary first-order axioms.

5.3 Reasoning in Attempto Controlled English

To support automatic reasoning in ACE we have developed the Attempto Reasoner (RACE). RACE proves that one ACE text is the logical consequence of another one, and gives a justification for the proof in ACE. Variations of the basic proof procedure permit query answering and consistency checking. Extending RACE by auxiliary first-order axioms and by evaluable functions we can perform complex deductions on ACE texts containing plurals and numbers.

Given the inconsistent ACE text

Every company that buys a standard machine gets a discount. A British company buys a standard machine. A French company buys a standard machine. There is no company that gets a discount.

RACE will determine two minimal unsatisfiable subsets

```
RACE proved that the sentence(s)
    Every company that buys a standard machine gets a discount.
    A British company buys a standard machine.
    There is no company that gets a discount.
are inconsistent.
RACE proved that the sentence(s)
    Every company that buys a standard machine gets a discount.
    A French company buys a standard machine.
    There is no company that gets a discount.
are inconsistent.
```

Given the ACE text

Every company that buys a machine gets a discount. Each of six Swiss companies buys a machine.

RACE can deduce the sentence

A company gets a discount.

as follows

```
RACE proved that the sentence(s)
    A company gets a discount.
can be deduced from the sentence(s)
    Every company that buys a machine gets a discount.
    Each of six Swiss companies buys a machine.
using the auxiliary axiom(s)
    (Ax. 9): Definition of proper_part_of.
    (Ax. 10-1): Every group consists of atomic parts.
    (Ax. 22-1): Number Axiom.
```

Note that this deduction uses predefined auxiliary first-order axioms that express domain-independent knowledge about the English language – for instance the relation between plurals and singulars – and about natural numbers – for instance their ordering relation. The axioms that pertain to natural numbers can also access evaluable functions expressed as Prolog predicates.

5.4 Future Work

Though ACE is already a powerful language supported by the tools of the Attempto system, much remains to be done to make ACE useful for the REWERSE community. In the following we briefly describe the requirements that we have identified so far. More tasks related to ACE and the Attempto system are described in the REWERSE Technical Annex.

5.4.1 Verbalisation of Formal Languages

The basic idea of the project Attempto is to replace formal languages by Attempto Controlled English (ACE) and to shield people who are not familiar with formal languages – and perhaps do not want to become familiar with them – from any trace of formality.

On the other hand we have to realise that formal knowledge representation languages are already widely used or – as in the case of the semantic web – are about to be used. How can we make documents expressed in RDF, OWL and other languages accessible to people who are not familiar with these languages? Our answer is "verbalisation of formal languages".

The ACE parser APE translates an ACE text into an equivalent discourse representation structure (DRS). In several project – unrelated to REWERSE – we have translated DRSs into various other formal languages, for instance into the standard language of first-order logic, into clauses, into the input language of a model generator, into the input language of an agent system, and into statements of a query language.

This experience suggests the reverse translation, i.e. the translation of formal languages equivalent to (a subset of) first-order logic into ACE via DRSs as common intermediate language. We call this verbalising formal languages.

Unfortunately, the parser APE cannot run backwards, i.e. generate ACE from a DRS. Thus we need a separate parser for generation.

We could rely on existing language development systems – for instance the KPML system[3] – to develop a parser for the translation of DRS into ACE. Experience shows, however, that independent parsers for the two translation directions will drift apart and eventually will no longer process the same languages. To prevent this problematic development, we plan to interleave the two parsers, that is to complement each grammar rule of APE for the direction ACE ⇒ DRS by an appropriate grammar rule for the direction DRS ⇒ ACE. Having the rules immediately adjacent will reduce the probability of the parsers drifting apart.

5.4.2 Support For Rule-Based Policy Specifications

The ACE language is perfectly suited to express rules. However, to serve as a rule language some elements are missing, namely

- labels, and

[3] cf. http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/README.html

- conflict resolution.

Grosof's Courteous Logic Programming [5] combines logic programming with prioritised conflict handling and classical negation, and still remains tractable. We plan to extend ACE by constructs that simulate Grosof's labelling and prioritised conflict handling.

5.4.3 Negation-As-Failure

For practical applications one often uses logic programs with negation-as-failure to implement a form of non-monotonic reasoning.

This is not sufficient, however, since some policy rules use both logical negation and negation-as-failure, as for example

If a customer did not transfer an amount of money and the bank cannot prove that he did not transfer the amount of money then the amount of money is credited to the account of the customer.

These two kinds of negation where independently proposed for Logic Programs by Gelfond and Lifschitz [3, 4] and by Pearce and Wagner [6, 8]. Later, Grosof developed Courteous Logic Programming [5] which makes use of both types of negation. Similarly, Woo and Lam [9] proposed stratified, extended logic programs providing both forms of negation.

ACE knows logical negation but not negation-as-failure. We plan to extend ACE by negation-as-failure, and we will take the proposals of Grosof and of Woo and Lam as guiding lines.

Grosof and independently Antoniou et al. [2] have shown that explicit negation-as-failure can be replaced by rules with priority relations. We will also investigate this approach.

5.4.4 Decidability

First-order logic is semi-decidable for satisfiability and entailment. For problems requiring the full expressiveness of first-order logic we have to live with this shortcoming, and - as experience shows - can often do so without negative consequences. Many practically occurring problems, however, are sufficiently constrained so that they can be expressed and solved in a decidable subset of first-order logic. Other problems, for instance those arising in safety-critical domains, even absolutely require decidability.

Of the many decidable subsets of first-order logic that have been identified so far, description logics and languages derived from description logics, for instance OWL DL, seem to be the most relevant in the context of REWERSE.

As the case of propositional logic shows, decidability does not necessarily imply computational tractability. Thus tractability is a separate issue. While OWL DL and its subset OWL Lite have been carefully designed to be tractable, their superset OWL Full comes without any computational guarantees.

Attempto Controlled English (ACE) – designed to provide high expressiveness – is equivalent to full first-order logic and thus semi-decidable. The question we plan to investigate is 'Which decidable, tractable and sufficiently expressive subsets does ACE have?'. Furthermore, these subsets must be clearly defined to be acceptable to the users of ACE.

Here are two approaches to find decidable, tractable and sufficiently expressive subsets of ACE.

In the first approach we restrict the syntax of ACE and then investigate the properties of the resulting sublanguage. A similar approach was chosen by Pratt-Hartmann [7] who defined a series of fragments of English and then for each fragment identified the computational complexity of determining satisfiability and entailment. Pratt-Hartmann's results are not very encouraging since his more expressive fragments of English have either exponential complexity or are undecidable.

The basic idea of the second approach is to map a suitable decidable and tractable subset of first-order logic, for instance OWL DL, to a subset of ACE. Before we can do this, however, we need to extend ACE at least by language constructs to describe classes and operations on classes. The mapping from first-order logic constructs to ACE constructs has to be done manually but could by supported by the Attempto verbalisation component (cf. "Verbalisation of Formal Languages" above).

Bibliography

[1] http://www.ics.mq.edu.au/ rolfs/controlled-natural-languages (controlled natural language homepage), 2004.

[2] G. Antoniou, M. J. Maher, and D. Billington. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming*, 42(1):47–57, 2000.

[3] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of Int. Conf. on Logic Programming*. MIT Press, 1990.

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

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

[6] D. Pearce and G. Wagner. Reasoning with negative information I – strong negation in logic programs. In M. K. L. Haaparanta and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*. Acta Philosophica Fennica 49, 1990.

[7] I. Pratt-Hartmann. Fragments of language. *Journal of Logic, Language and Information*, (13):207–223, 2004.

[8] G. Wagner. A database needs two kinds of negation. In B. Thalheim and H.-D. Gerhardt, editors, *Proc. of the 3rd. Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag, 1991.

[9] 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.

Part II

Requirements and Scenarios

Chapter 6

Reference scenarios

6.1 A European use case on financial services

Background: This use case for European financial services was originally presented by Steve Ross-Talbot; it involves Basel 2 [1], which is a major initiative in Europe to enforce policy within the financial services domain. This includes retail and wholesale banking as well as insurance services. The aim of Basel 2 is to ensure that the capital adequacy is properly enforced across all financial service transactions within Europe. This includes everything from pensions to insurance and mortgages.

As we move to a Europe which is free of barriers to trade we shall see more and more cross border selling of financial service products. These products and services range from mortgages to insurance to general savings products. These products underpin the fabric of European life from secure accommodation for European citizens, to pensions. Further, it secures real inward investment in European industry through equity participation and through European wide fixed income products.

In Europe today it is not made easy for consumers to purchase such products nor is it made easy for consumers to manage those products through their life cycle. Initiatives in member states are trying to address some of the issues as they pertain to the entire life cycle of financial service products (e.g. Britain's FSA's[1] CP98 [2], CP121 [3], CP136 [4]). In Europe many of the same issues are being addressed by Basel 2. The result will be a regulated and safe environment in which products can be bought and sold and in which a fair market can be ensured through the semantic comparisons necessary to support fair trade.

It is against this landscape and these regulatory requirements that we set out the use case to show the relevance of rules and the power of the Semantic Web in meeting the needs of European citizens and empowering them through the web to continue to purchase and manage European financial service products.

Scenario: Our scenario involves buying a simple mortgage package that includes some level of insurance that is used to underpin the risks inherent in products that have a very long lifetime, while leveraging a rules-based Semantic Web.

Juan Sebastien is moving from Madrid, having got a new job in Kaiserslautern. He would like to buy an apartment in Kaiserslautern for him and his family. He is married with one

[1] http://www.fsa.gov.uk/

child and his wife is expecting their second in about 6 months. Before he leaves he wants to buy a mortgage package (mortgage, property insurance and life insurance) that suits his needs. Being a citizen of Europe he would like to buy from Europe and is seeking the best solution. Today he can barely identify the best solution, since there is no way for him to easily compare products which have a varied set of add-ons or give-back clauses attached to them, or even be aware and understand them.

The Semantic Web of the future will provide Juan Sebastien with the necessary self-support to buy the products that he wants over the web. It will provide him with the ability to compare products through the use of rule based agents and ontologies, the underpinning of the Semantic Web. Juan Sebastien, in this brave new world, will be able to ask questions of the Semantic Web that he has never been able to ask of the web before, such as,

- What is the cheapest product?
- Why do I need to provide a 20% deposit for this product?
- Why does Bank X deny my request for product Y?
- If I am locally employed, but without citizenship, yet living more than 3 years there, can I still get the special discount "as advertised"?

Furthermore, the back-office functions can be guaranteed to ensure that policy, as it pertains to Basel 2 is adhered to, and that Juan Sebastien's rights, as they pertain to freedom of information (*why not* as well as *why*), are maintained.

Another typical complication is if Juan's company is an international company, with global rules for employee's eligibilities, and with added local variation as necessary.

We propose Rule-Based Semantic Web technology to provide a higher level of self-support in such scenarios: Rule-Based Semantic Web technology is a precursor – its ontological modeling being of paramount importance to the ability to compare concepts – to the reasoning, compliance as well as positive and negative explanations for decisions that are made. The rules will ensure the consistency of the policy, easy sharing (even across national boundaries), and of course, easy update.

Novelty of the approach

- European Business Language
  - Rules are in the common business language of the provider
  - Questions/explanation are in the language of the consumer
  - Markup techniques (based on an XML language like RuleML) for rule exchange between providers and between agents on the web
- User Empowerment
  - Domain specific templates and natural language techniques to ensure user's independence in rules maintenance
  - Knowledge sharing, rules visibility and easy exchange of ideas as part of the Semantic Web
- Advanced queries and reasoning

  - Why and why not queries
  - What-if queries

- Enhanced Productivity
  - A single specification for business rules *enforcement* and advanced *queries*

6.2 Privacy protection

Background: New generation access control methods for open systems are based on the exchange of electronic credentials and declarations (licence agreement acceptance, personal data, etc.). Fine grained access permissions are likely to require numerous such exchanges during each service use. Moreover, as users navigate across multiple sites or request composite services (activating other component services), the number of credential and declaration requests may grow enough to deteriorate usability and discourage the use of protected resources.

A second, related issue is privacy protection. Electronic credentials and declarations may disclose sensitive information about the users. Several national regulations are imposing limitations to the use and distribution of sensitive personal data. Still, users may want to reduce the amount of sensitive information they distribute to servers in order to prevent abuses.

Then, suitable software components called *personal assistants* (PA) are meant to handle credential exchange. Personal assistants are assigned two important tasks: making credential exchange as transparent as possible while observing the user's distribution policy.

Scenario: Maria Rossi is connected to NewAgePortal. The server wants Maria Rossi to provide her personal data before granting access to the portal's services. Maria Rossi's PA asks NewAgePortal for its privacy policy (e.g., formulated with the P3P standard), and compares it with Maria Rossi's policy. The latter is only partially fulfilled by the former, so the PA sends to NewAgePortal a form which is only partially compiled, possibly using imprecise data (e.g., only birth year instead of full birthdate, or a zip code instead of full address). If NewAgePortal accepts the partially compiled form as valid data, then navigation proceeds. Otherwise, the PA may try different strategies, for example:

- if the user's policy allows a distributed trust model, then the PA may check whether NewAgePortal is commonly regarded as a reliable service; if so, the PA may send a complete form to the server;
- the PA may ask the user whether she wants to make an exception for this site; for this purpose, the PA may have to explain some of the features of the server's policy, highlighting how much it fulfills or departs from the user's policy;
- if the user's policy allows for lies, then the PA may fill in the form with incorrect information; this strategy may be appropriate for some fields only, e.g., a false address may cause delivery of a purchased article to fail; in some other contexts, a wrong address may have no drawbacks.

One complication concerns *usability*. Given a service's access control policy and a user policy, is there any means for the client to get the service? Given a library of standard user policies and a service policy, it may be interesting to perform this kind of analysis automatically, before the service – or a new policy for that service – is activated.

Further complications concern policy *composition*. The policy of any given organization may have to be merged with the current regulations about privacy. The goal is synthesizing a compound policy that enforces both the organization's policy and privacy protection in a *maximally cooperative way*, that is, with a minimal number of restrictions and constraints.

Novel aspects:

- Advanced trust models and trust management
- Automated policy comparison
- Power to the user
  - user preferences
  - interactive policies
  - domain specific templates and natural language techniques to ensure user's independence in rules maintenance
- (semi) Automated policy composition
- (semi) Automated policy validation

6.3 Inter-organization business processes

Scenario: The architecture studio Bauhome has to interact with the offices of the city of Munich to get authorizations, update building descriptions in the official databases, etc. The city of Munich exports semantic workflow descriptions of all these processes, specifying which forms are needed, whom they must be addressed to, what prerequisites should be fulfilled by the requestor, etc. Such descriptions may change along time, as regulations and internal organization change. Similarly, Bauhome describes its own internal workflows through machine-understandable rules and formats.

From such descriptions, and given goals like "Construct new building in area X", "Restore building Y", etc., Bauhome's computer system automatically sets up an appropriate workflow which is then fed into a workflow management system to support the goal activities. Changes to regulations or office processes are automatically reflected into the workflow by re-planning the activities using the new process descriptions and rules. Similarly, unexpected changes to the internal organization of Bauhome may cause the workflow to be adapted to the new situation in a machine-assisted way.

6.4 Sample policy verbalizations

Nonmonotonic rules: Nonmonotonic rules and nonmonotonic negation (negation as failure) are essential to make decisions in the absence of complete information or complete directives. Very common examples include:

- *Open policies*: In the absence of explicit denials, access is granted;
- *Closed policies*: In the absence of explicit authorizations, access is denied.

- *Inheritance*: User, object and action *hierarchies* are used as a means for expressing policies in a concise and manageable way.

  For example, one may say that by default administration employees may read the folder `Admin`. This rule is propagated down the hierarchy of users to all the members of the administration, including – say – Joe Jackson; moreover the rule is propagated down the hierarchy of objects to all the files in the subtree of the file system below the directory `Admin`, including – say – `letter.pdf`. Then Joe Jackson is allowed to read `letter.pdf`.

  Then it should be possible to formulate exceptions, such as "Joe Jackson cannot read `letter.pdf`", and adopt a suitable form of overriding to handle this apparent inconsistency.

The issue here is how to verbalize nonmonotonic rules, that is, how to formulate them in controlled natural language. A rule triggered by the non-derivability of an authorization can be formulated in a number of alternative ways, e.g.:

- if nobody said that X can .... then ...
- if no authorization says that X can ... then ...
- if the policy does not state that X can ... then
- if there is no authorization "X can ..." then ...
- X can ... if not explicitly forbidden
- X can ... unless stated otherwise
- if no user is authorized to ... then ...
- if X is not given the authorization to... then ...

Policy composition: (meta-rules) Given two policies P and Q (possibly crafted by different (sub-) organizations, it should be possible to express compound policies such as:

- Grant the least/maximal privilege among those granted by P and Q
- As far as topic X is concerned, override P with Q
- Remove from P the authorizations occurring in Q
- Restrict P to object classes $O_1, \ldots, O_n$
- Close policy P under the rules in Q

Remark: From the above discussion it seems that nonmonotonic rule formulation and policy composition require linguistic means to refer to entire policies *within the policies themselves*.

Open issues:

The appropriate way of verbalizing the following specifications still needs to be found:

- Boolean conditions on file and user attributes.

- Recursive conditions (e.g., for defining certificate chains)

  Certificate chains are needed to specify chains of trust. Technically, certificate chains are sequences of certificates $C_1, \ldots, C_n$ such that each certificate C_i $(1 \leq i < n)$ certifies the public key of the issuer of C_{i+1}. The issuer of C_1 should be a known certification authority.

- Default authorizations for classes of users, objects, and operations.

  In particular, how to select different forms of overriding (most specific takes precedence, explicit preferences attached to the rules, most specific along a path takes precedence, non-overridable authorizations – also called strong and weak authorizations –, etc.)

Bibliography

[1] Basel Committee. Basel II Revised International Capital Framework, http://www.bis.org/publ/bcbsca.htm, 2004.

[2] Financial Services Authority. CP98: The Draft Mortgage Sourcebook, http://www.fsa.gov.uk/pubs/cp/98, 2001.

[3] Financial Services Authority. CP121: Reforming Polarisation: Making the market work for consumers, http://www.fsa.gov.uk/pubs/cp/121, 2002.

[4] Financial Services Authority. CP136: Individual Capital Adequacy Standards, http://www.fsa.gov.uk/pubs/cp/136, 2002.

Acknowledgments

We thank Marianne Winslett from the Dept. of Computer Science of the University of Illinois at Urbana-Champaign for her contribution to "Trust Negotiation" in the section "Policy-based trust management" of Chapter 2, and we thank Steve Ross-Talbot for the use case for financial services presented in Chapter 6.

We also thank Gerd Wagner and our internal reviewers for the valuable feedback they provided, which helped us improving the quality of this report.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`)