I5-D1

State-of-the-art on evolution and reactivity

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/Lisbon/I5-D1/D/PU/a1
Responsible editor(s):	José Júlio Alferes and Wolfgang May
Reviewer(s):	Thomas Eiter
Contributing participants:	Dresden, Eindhoven, Goettingen, Lisbon, Munich, Skoevde, Melbourne
Contributing workpackages:	I5
Contractual date of delivery:	31 August 2004

Abstract
This report starts by, in Chapter 1, outlining aspects of querying and updating resources on the Web and on the Semantic Web, including the development of query and update languages to be carried out within the REWERSE project.

From this outline, it becomes clear that several existing research areas and topics are of interest for this work in REWERSE. In the remainder of this report we further present state of the art surveys in a selection of such areas and topics. More precisely: in Chapter 2 we give an overview of logics for reasoning about state change and updates; Chapter 3 is devoted to briefly describing existing update languages for the Web, and also for updating logic programs; in Chapter 4 event-condition-action rules, both in the context of active database systems and in the context of semistructured data, are surveyed; in Chapter 5 we give an overview of some relevant rule-based agents frameworks.

Keyword List
Logics and languages for updates, ECA rules, active databases, rule based agents

State-of-the-art on evolution and reactivity

José Júlio Alferes[1], James Bailey[2], Mikael Berndtsson[3], François Bry[4], Jens Dietrich[5], Alexander Kozlenkov[6], Wolfgang May[7], Paula Lavinia Pătrânjan[4], Alexandre Pinto[1], Michael Schroeder[8] and Gerd Wagner[9]

[1] Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa

[2] Department of Computer Science and Software Engineering, The University of Melbourne

[3] School of Humanities and Informatics, University of Skövde

[4] Institut für Informatik, Ludwig-Maximilians-Universität München

[5] Institute of Information Sciences and Technology, Massey University

[6] Department of Computing, City University, London

[7] Institut für Informatik, Universität Göttingen

[8] Biotec/Dept. of Computing, TU Dresden

[9] Faculty of Technology Management, I&T, Eindhoven University of Technology

5 August 2004

Abstract

This report starts by, in Chapter 1, outlining aspects of querying and updating resources on the Web and on the Semantic Web, including the development of query and update languages to be carried out within the REWERSE project.

From this outline, it becomes clear that several existing research areas and topics are of interest for this work in REWERSE. In the remainder of this report we further present state of the art surveys in a selection of such areas and topics. More precisely: in Chapter 2 we give an overview of logics for reasoning about state change and updates; Chapter 3 is devoted to briefly describing existing update languages for the Web, and also for updating logic programs; in Chapter 4 event-condition-action rules, both in the context of active database systems and in the context of semistructured data, are surveyed; in Chapter 5 we give an overview of some relevant rule-based agents frameworks.

Keyword List

Logics and languages for updates, ECA rules, active databases, rule based agents

Contents

Chapter 1

Towards evolution and reactivity on the Web

Use of the *Web* today –commonly known as the "World Wide Web"– mostly focuses on the page-oriented perspective: most of the Web consists of browsable HTML pages only. From this point of view, the Web can be seen as a graph that consists of the resources as nodes, and the *hyperlinks* form the edges as it is e.g. done in the notion of a Web skeleton of HTML pages and their hyperlinks in [ML04]. Here, queries are stated against individual nodes, or against several nodes, with formalisms such as F-Logic [LHL+98] or Lixto [BFG01]; or in case that the sources are provided in XHTML, they can be queried by XQuery, XPathLog [May01b], or Xcerpt [BS02]. Data extraction from the Web lead by the idea of extracting knowledge based on browsing has been investigated e.g. in [LHL+98] or [ML04]. As such, the Web is mainly seen from its static perspective of autonomous plain data *sources*, whereas the *semantics* and the *behavior* of the sources, including active interaction of resources, does not play here any important role.

But there is more on the Web of today than HTML pages. Leaving the superficial point of view of HTML pages, the Web can be seen as a set of *data sources*, some of which are still browsing-oriented, but there are also database-like resources that can actually be queried. In fact, besides HTML documents that are accessed as a whole, and in which (query) processing has then to be done separately, there are XML (including XHTML) data sources, described by DTDs or XML Schemas, that in general can be queried (e.g., by XPath/XQuery, XPathLog or Xcerpt). Moreover, specialized information sources (that we abstractly call *Web Services*) exist that provide answers only to a restricted set of queries that are given abstractly as a set of name-value pairs[1]. We refer to the Web as a collection of "plain" data sources of this kind as the **Conventional Web**, populated by (interlinked) HTML, XML, DTDs, and XML Schema (either as files, or as databases or Web Services that "speak" HTML/XML).

With these representations, the Web could be seen as a graph of data sources that can be queried as a whole. In the last years, plenty of research on data integration on the Web has been done [Len02, Len03, Hal03], mainly using view-based mappings between data sources.

[1] For the sake of simplicity, in the whole of the document we will refer to abstract nodes in the Web (graph) simply as information resources or data sources though, as it will become clear, we are not thinking of these nodes as "simply" providing information and sometimes they are, e.g., Web services. For a discussion on the distinction between information sources and Web services see e.g. [LW00].

Continuing this process, the perspective shifted further to the idea of a Web consisting of (a graph of) *information systems*. With these information systems, data extraction may be thought not only in terms of local queries, but also in terms of global queries that are stated against "the Web", or against a group (or community) of nodes on the Web (sharing a common application area and ontology; often, a community is represented by a *portal*), and to *intelligent, semantic* capabilities dealing with the available information. Given the highly heterogeneous and autonomous characteristics of the Web, this requires appropriate query languages, and a way to deal with the integration of data from the various sources.

Such an infrastructure of autonomous, cooperative resources will allow for more than plain querying. The questions in the design of this **Semantic Web** are e.g. what functionality is needed, how should it be implemented and, especially, what communication structures are required. We will discuss these issues in the following, in general, and especially from the point of view of *evolution and reactivity*. Let's start with an example:

Example 1.1 (Introductory Example: Travel Agency) *Consider a set of resources of travel agencies and airline companies. It is important to be capable of querying such a set for, e.g. timetables of flights, availability of flight tickets, etc. But a Web consisting of information systems should allow for more. For example: it should allow for drawing conclusions based on knowledge (e.g. in the form of derivation rules) available on each node; it should allow for making reservations via a travel agency, and automatically make the corresponding airline company (and also other travel agencies) aware of that; it should allow airline companies to change their selling policies, and make travel agencies automatically aware of those changes; etc.*

The Semantic Web, as we see it, with such capabilities can be regarded as forming an active, "living" infrastructure of autonomous systems. This view of the Web raises new issues, and requires new concepts and languages, that we intend to develop within the REWERSE – *Reasoning on the Web with Rules and Semantics* – project.

In the present deliverable that represents the starting point of *Working Group I5, "Evolution and Reactivity"*, we focus on the aspects –that will be outlined in the remainder of this chapter– that are relevant for this topic. It will be clear from this outline that several existing research areas and topics are of importance for our proposed endeavor. In the following chapters we make a systematic overview of the state-of-the-art in a selection of such areas and topics with particular emphasis on the aspects that are more relevant for the intended work:

Though this deliverable (and working group) is not directly concerned with the subject of query languages, that is studied and developed elsewhere in REWERSE, these are of importance for evolution and reactivity (albeit because update languages are usually based on query languages) and, as such, we briefly discuss query languages for the Web in Section 1.1. Then, in Sections 1.2 to 1.5 we discuss dynamical aspects, languages for reactivity and evolution, and communication on the Web. Reasoning on the Semantic Web, especially in what concerns the dynamic aspects, are then briefly discussed in Section 1.6.

1.1 Web Query Languages

1.1.1 Today's Web Query Languages

Query languages similar to, but different from, those used and developed for (XML) databases are needed on the Web for easing the retrieval of data on the Web, or easing the specification

of materialized or non-materialized views (in the database sense) and for expressing complex updates, especially intentional updates (i.e., updates of intensional predicates expressed in terms of queries) that must then be translated into updates of the actual sources.

Queries play an important role in the Web, beyond user-defined queries alone: dynamic Web pages (i.e., those whose content is generated at query time) are non-materialized views (i.e., views that are computed at data retrieval time). Here, closedness of the language is obviously necessary: it is defined over Web contents, and its output must be valid Web contents.

Design of Web Query Languages. Most of the early Web query languages developed from query languages for semistructured data, and have then been migrated to XML. The design decisions and experiences of these languages have to be considered when designing query and manipulation languages for the Web as a whole. An overview of these can be found in [May04].

Logic programming-style rule-based languages (immediately including an incremental update language) have been presented with e.g. *WSL/MSL (Wrapper/Mediator Specification Language)* [GMPQ+97], and F-Logic [LHL+98]. *Lorel* [MAG+97] and StruQL [FFK+97], following the SQL/OQL-style clause-based design; StruQL was then developed into XML-QL [DFF+98]. Other languages developed from the area of tree matching and transformations, e.g., *UnQL* [BDHS96], *XDuce* [HP00], or *YATL* [CDSS99].

The standard XML languages developed from the experience with the above-mentioned languages and from the SGML area. XPath [xpa99] has been established as an addressing language. It is based on path expressions for navigation, extended with filters. It serves as the basis for many other W3C languages in the XML world. XQuery [xqu01] extends XPath with SQL-like clause-based constructs `FLWOR` (`FOR ... LET ... WHERE ... ORDER BY ... RETURN`) to a full-fledged query language. Variables are bound in the `FOR` and `LET` clauses to the answer sets of XPath expressions. The `WHERE` clause expresses further conditions, and the `RETURN` clause creates an XML structure as a result. While the selection parts in the `FOR` and `LET` clauses are XPath-based, an XML pattern in XML-QL style is used in the `RETURN` clause where the variables are embedded into literal XML and constructors. Proposals for extending XQuery with update constructs have been published in [TIHW01, Leh01]. As a *transformation* language, XSLT [xsl99] follows a completely different idea for providing information from the Web: In it, transformation rules specify how information is extracted from a data source.

Recent non-Standard XML Query Languages. The Lixto system [BFG01] uses a graphical interface for querying HTML, where a query is developed by interactively selecting nodes from a browser presentation. Internally, Lixto uses a logic programming language for XML called *Elog*, based on flattening XML data into Datalog. XPathLog [May01b] combines (i) first-order logic, and (ii) XPath expressions extended with variable bindings. It is completely XPath-based (i.e., navigational access that is extended to Datalog-like patterns) both in the rule body (query part) and in the rule read (update part), thereby defining an incremental update semantics for XPath expressions. *Xcerpt* [BS02] is a *pattern-based* language for querying and transforming XML data. Its basic form follows a clean, rule-based design where the query (matching) part in the body is separated from the generation part in the rule head. XML instances are regarded as terms that are matched by a term pattern in the rule body, generating variable bindings. The embedding of Xcerpt terms into `CONSTRUCT ... FROM ...` clauses allows for nesting. An extension with updates, called *XChange*, is currently being designed as a clause-based language. An overview of proposals of update languages for XML and the Web can be found in Section 3.1.

Comparison of Design Concepts. The existing languages for handling semi-structured data and XML differ in several facets in terms of the concepts they use. For example: access

mechanisms and homogeneity of the query and generation part (patterns, path expressions); underlying data model (graph vs. tree); nature of the underlying theoretical framework (logic-based or denotational semantics); and last but not least clause-based or logic-programming-style syntax and semantics.

All languages discussed above are *rule-based* and *declarative*, generating variable bindings by a matching/selection part in the "rule body" and then using these bindings in the "rule head" for generating output or updating the database. This rule-based nature may be more or may be less explicit: F-Logic, MSL/WSL (Tsimmis), XPathLog, and Elog use the ":-" Prolog syntax, whereas UnQL, Lorel, StruQL, XML-QL, XQuery, and Xcerpt couch their rule-like structure in an SQL-like clause syntax. These clausal languages allow for a straightforward extension with update constructs.

1.1.2 Requirements on Query Languages for the Web

Obviously, a Web query language must allow for the explicitly inclusion of multiple data sources from the Web. While this was no issue for SQL, all above XML languages provide this functionality by using the URL, mechanism (which is self-evident for those that emanated from HTML Web data extraction), e.g., by adopting XPath's document() function. Note that the smooth extension from a database query language to an update language does not necessarily carry over to Web query languages: this would require separate additional communication, and data that is publicly accessible is far from being publicly updatable.

Coming back to queries on the Web, these should also be robust against certain changes in a data source, e.g., splitting a data source over several ones that are linked by XLink/XPointer, or wrapping parts of an XML data source in a Web service (which makes querying immediately also an issue of reactive behavior and communication). In such cases, the execution model must be modular in the sense that it transparently combines answers contributed by multiple data sources. It must also consider that data sources provide different, possibly restricted, functionality like answering arbitrary queries, providing access to bulk data, answering a fixed set of predefined queries. Thus, closedness is also an important property: query languages are used to define views that act themselves as virtual data sources, and that must be in the same format as the other sources.

In the context of the *Semantic* Web, querying goes much beyond than "only" accessing the base data, as can e.g. be done by SQL or by XML query languages. Query answering on the Semantic Web in general also requires the use of metadata expressed , e.g., using RDF/RDFS or OWL, but maybe also thesauri (as used in information retrieval and computational linguistics), and thus must support *reasoning* about the query, data, and metadata. The actual forms of reasoning to be applied with RDF/RDFS, OWL, thesauri, etc., and in general on the forthcoming Semantic Web are rather unclear today – this is one of the goals of current research of REWERSE.

For providing a base for such research, flexible and extensible query languages including "lightweight reasoning facilities" appear to be a good choice for a first step towards generic tools for the Semantic Web. For several reasons, we propose to use rule-based languages for querying.

Rules provide a natural modularization, and the rule-based paradigm covers a lot of sub-paradigms for designing a wide range of language types as described above: SQL and XQuery are actually a form of simple rules (for defining views); XSLT is rule-based; and languages like Datalog provide a multitude of semantics for any kinds of reasoning. Coming back to the

modularization issue, a simple form of non-monotonic negation in presence of recursion (such as restriction to stratified programs) seems to be sufficient for a deductive query language for the Semantic Web. The expressive power can be restricted and extended by choosing suitable semantics – e.g. recursion is needed for computing closure relations (as those of RDF and OWL) and can be easily "bought" with the use of a fixpoint semantics. Recursive rules are a convenient way to express common forms of reasoning on the Semantic Web, e.g. for traversing several edges relating remote concepts. Such complex semantics of rule languages are well investigated in logic programming. In [BFPS04a] it is shown how the rule-based language Xcerpt can be used for reasoning on Web metadata such as RDF and OWL data. Reasoning about metadata, classes and default properties is e.g. provided by XPathLog/LoPiX [May01c] that integrates the F-Logic/Florid mechanisms [KL89] for handling metadata and reasoning (signature, classes, subclasses and nonmonotonic inheritance).

Additionally, rule-based languages can easily be extended to updates and in general to the specification of dynamics on the Web. Here, research on event-condition-action rules and active databases can be applied and extended. As another consequence of their modularity, rule-based languages are relatively easy to understand and program. The use of rule-based languages allows also for a modular design of language processors. These consist then of two components: one for the interpretation of rule heads and bodies, and one for implementing the global semantics for suitably evaluating a set of rules.

The language base will most likely be provided by XML, where the individual languages are distinguished by their syntax, i.e., their namespaces, and their element and attribute names. Thus, a generic rule processor will be able to handle arbitrary rules (not only query rules, but also updates and general reaction rules) and to apply appropriate processing to each type of rules. This way, new languages too can be integrated at any time.

1.2 Local Behavior: Answering Queries and Being Reactive

The most basic and primitive activity on the Web is query/answering. From the point of view of the user, querying is a static issue: there is no actual dynamic aspect in it (except possibly a delay). Nevertheless, from the point of view of the resources, there comes reactivity into play: when answering queries, the resources must answer in reaction to a query message, and in case that the query is answered with cooperation of several resources, they must also send messages to other resources.

Such cooperation is in general required when considering communities of resources on the Web. Even more, cooperation and communication is already necessary when a resource contains references in any form to another resources. Two types of cooperation for query answering can be distinguished:

Distributed Query Answering of Explicit Queries. Explicit queries in any XML query language can be stated against XML resources whose external schema is known. Such resources can contain references to other XML resources by XLink elements with XPointers. A model for answering queries in the presence of XLink references has been investigated in [May02]. The actual evaluation of a query in this model results in (re)active communication of result sets and/or subqueries between resources. This behavior is a simple form of reactivity.

Query Answering by Web Services. Web services can also be used for answering (often only a specific set of) queries. For this, they can either use their local knowledge (e.g., facts and

derivation rules), or they can contact other resources for answering a query (e.g., a travel agency service that uses the schedules of trains and flights from several autonomous resources) as made possible by e.g. WSDL/OWL-S/UDDI [wsd01, OWL03, Uni02] (finding an appropriate service on the Web), or, as with Active XML [ABM+02] (using a known service; see also Section 4.2.2), they can contain embedded calls in their local data, that when fired enrich the local data with the call's results. In the latter cases, the behavior of the Web service is actually not simply query-answering, but can be seen as a general (re)active behavior, according to an external interface description that can provide much more functionality than only query/answering. Such Web Services are instances of general *reactivity*: there is a message, and the resource performs some actions. Moreover, Web Services may have an internal state (e.g., reservation systems) that they update.

For including general reactivity of Web Services into the intended reasoning on the Semantic Web, a formal specification of their behavior is required. Here, we distinguish two types of Web Services, according to the possibilities of reasoning about their behavior (and updating it): Web Services with rule-based behavior (including queries), and Web Services whose behavior is described STRIPS-like [FN71], or with another similar formal method. Other Web Services will be considered as black boxes.

Note that, so far, we have described a mainly non-changing, but *active* Web consisting of nearly isolated passive and only *locally* active Web Resources. In this setting, evolution on the Web takes place by: either local changes in data sources via updates (e.g. changing a flight schedule); or local evolution of Web Services by reaction to events due to their own, specified, behavior. However, this scenario already raises a number of issues deserving investigation, and will serve as our base case for *reactivity*.

Global Aspects of Distributed Query Answering in the Web. From this "simple" base case – cooperative answering of queries by "the Web", independent different dimensions can be derived:

- The definition of a global model theory and reasoning about global constraints and consistency aspects based on computing answers to queries.
  Note that at this point, we do not yet have to deal with mediation and mapping between different data sources – still, only the base data with explicit links (that explicitly incorporate potential mappings) is considered;
- Basic peer-to-peer communication between resources via *messages* and simple *reaction rules* for answering queries;
- Extension from queries to updates (where updates require the evaluation of a query e.g. for addressing the actual item to be updated). Such remote updates on the XML level will provide our simplest case of *dynamics*. Here, in addition to materializing a remote update, access control and authorization has to be considered. Research in this area is done by Rewerse's *WG I2: Policies*.

1.3 Updates and Evolution of the Web

Evolution of the Web is a twofold aspect: on today's Web, evolution means mainly evolution of individual Web sites that are updated locally (see discussion of remote updates above). In contrast, considering the Web as a "living organism" that *consists* of autonomous data sources, but that will *show* a global "behavior" (as already discussed above for query answering) leads

to a notion of evolution of the Web as *cooperative evolution* of (the state of) its individual resources.

The state of such a system could be classified into three conceptual levels: facts (e.g. in XML or wrapped in XML); a knowledge base, given by derivation rules (e.g. marked up in RuleML); and behavior (e.g., reaction rules[2] specifying which actions are to be taken upon which event and given what conditions).

In this context, update concepts are needed for several tasks, besides the updating of the contents of a Web site by its owner. It should also be possible to update the contents of a Web site as a reaction to actions of some user (e.g. when a user books a flight, the reservation of a seat must be stored in the underlying database). This can be either an update of the local database of the respective Web site but, most likely, this is an update to a remote database (e.g. in the above example when the booking is done via a travel agency, and the reservation must be forwarded to an update of the carrier company). This also requires the definition of ways to communicate updates.

Moreover, update languages must also allow for updating the derivation rules and the behavior of a Web site, e.g. in adaptive Web systems, where the user modelling data must be updated, and the behavior of the system will change. Updates of the latter type are mostly executed by the owner, but it should also be possible to receive updates of such types from (authorized) external partners. For example, a website that provides tax consulting should be updatable from a central service if the taxing rules change.

Example 1.2 *Consider a Web site of an online shop, where for all articles the prices without and with VAT are given. An update of the VAT rate to this page would require updating the prices of* all *items of an answer set with an individually computed new price. Obviously, a good design in reality would only store the net price and compute the gross price by multiplying it with the current percentage for the VAT.*

Here, the update can take place in several ways: (i) the owner of the page regularly polls the current VAT rate and updates its page accordingly; (ii) the owner subscribes some service that informs him in case the VAT changes; or (iii) the owner tells an appropriate service to invoke an update on its data (either to the VAT rate data, or to the rule that computes the gross price) in case the VAT rate changes. The latter case can be implemented either by an explicit update statement, or by an update on the semantic level, i.e., "(VAT, is, 20%)" in RDF or as a rule (suitably markedup) "compute the gross price by adding 20% VAT on the net price" that is then translated into the local derivation rule language (or, the service uses the "global" rule language directly as its internal rule language).

This example also illustrates the close connection between "update", "reactivity" and "evolution" of a Web site.

When updating the Conventional Web, the update is expressed as a specific update operation on a specific Web site (in general, updating facts on the data level). When derived data, whose base data is taken from another site, is "updated" appropriate actions must be taken – view update, requiring again appropriate communication. On the other hand, if another Web site has ever used the updated data (e.g. providing a materialized view), the update must be accordingly propagated – view updating, again including communication means.

For the Semantic Web, updates should be expressible on a semantic level, requiring a declarative, semantic framework for generically specifying the update, and appropriate mechanisms in the receiving Web site to materialize the update.

[2]For a discussion on these reaction rules, see Section 1.4.

Local update languages, that are surveyed in Section 3.1, are in general extensions of query languages: the query language is used (i) for determining what data is updated, and (ii) for determining the new value. Note that in both cases, the respective answer is in general not a single data record or a single value, but can be a set or – in the XML case – a substructure. Moreover, the value of (ii) can be dependent on the current item of the answer computed in (i). Update languages for the *Semantic Web* are to be based on query languages for the Semantic Web, e.g., for RDF.

Thus, having data flow and dependencies between Web sites, besides a plain communication mechanism, but mechanisms to maintain consistency between Web sites by update propagation are required. Update propagation consists of (i) propagating an update, and (ii) processing/materializing the update at another Web resource. The latter, we have just seen, is solved by local update languages. So the remaining problem turns out how to communicate changes on the (Semantic) Web. Often, a change is not propagated as an explicit update, but there must be "evolution" of "the Web" as a consequence of a change to some information.

Example 1.3 (Update on the Semantic Web) *Consider an update in the timetable of Lufthansa that adds a daily flight from Berlin to Timbuktu at 12:00, arriving at 20:00. There will be several other resources, possibly using different ontologies and belonging to different communities, for whom this information is relevant. First there are German travel agencies etc. who use the Lufthansa schedule for their offers. For these, it is probably the case that they are directly informed, or that they query the LH timetable regularly.*

There will also be other resources that have no immediate connection to the LH resource. E.g., a taxi service at Timbuktu can have the rule "when a European airline provides a flight to Timbuktu, we should provide a car at the airport with a driver who speaks the language of the country where the plane comes from". Here, the resource should somehow be able to detect updates somewhere on the Semantic Web that are relevant for itself.

For this, it is clear that we need a "global" language for communicating changes, and communication strategies how to propagate pieces of information through the Semantic Web, seeing it globally as the union of *all* information throughout the Web. In it, the *Semantic*-property of the Web is crucial for automatically mediating between several actual schemata.

This problem again is twofold: (i) infrastructure and organization, and (ii) language. For the former, e.g., *polling (pull)*, *broadcast (push)*, *publish-and-subscribe (push+pull)* (see e.g. [TRP+04]) and *continuous query services (pull+push)* (see e.g. [CdTW00]) can be used. A discussion on infrastructure and organization can be found later, in Section 1.5. Next, we investigate language aspects of this communication.

1.4 Communication on the Web: Reactivity, Events and Actions

We assume that communication in the Semantic Web takes place by *peer-to-peer* communication between resources (see also Section 1.5). Furthermore, this communication is based on *messages* and *events* (that both are represented in XML). Following a well-known and successful paradigm, we propose to use rules, more specifically, *reactive rules* similar to *Event-Condition-Action (ECA)* rules for communication and for specification of the local behavior of Semantic Web nodes. Several languages with reaction rules can be found in the literature. For an overview

of ECA rules in the context of active database systems and Web languages see Chapter 4, and for an agent's language with reaction rules see Section 5.1.

An important advantage of ECA rules is, again, that the *content* of the communication can be separated from the *semantics* of the rules themselves. Communication is then done by raising/signalling an event. The depending resources process this event (either it is delivered explicitly to them, or they poll it via the communication means of the Web – see Section 1.5), then optionally check a condition (that e.g. tests if the update is relevant, trustable etc.), and finally take an appropriate action (in most cases, updating their own information accordingly).

Events are either atomic events (e.g., updates in the local databases, or received messages), or complex events formed as combinations of atomic ones (e.g. "when A is updated, and then B is updated") expressed in some event algebra; conditions denote queries to one or several nodes and are to be expressed in the proposed query language; atomic actions are e.g. update requests or sending a message, and actions can be grouped as transactions. Such a transaction could for example express "book a flight and then, if a flight has been found, book a hotel at the arrival place" (the ACID properties of the transaction ensure that either all nor nothing of this is done).

Accordingly, the language for these rules must comprise the language for events (in the "Event" part), the language for queries (in the "Condition" part), and the language for updates (in the "Action" part). The language will be based on XML, using a sublanguage for rule markup, an event sublanguage, and an action sublanguage. Consequently, also the query language will have an XML representation (which would e.g. also allow to map queries between different query syntaxes, e.g., XQuery and Xcerpt).

1.4.1 Events

As for the "Event" part of the rules, languages for working with and describing events, and corresponding processors of events, are necessary. The design of event languages should be based on the experiences in the area of *active database systems*. Since reactivity is based on reacting to events, our required notion of an event must be quite general and amount to almost any detectable occurrence on the Web.

Traditional active rule-based behavior (e.g., in active databases – see Chapter 4) deals only with local, explicit events that are easy to detect. On the Web, *remote* events are often also relevant, e.g. in case that a Web site provides a materialized view of remote data. In the Conventional Web, these are explicit changes on data, where a (simple) query can be given for selecting this piece of information. In the Semantic Web, these can be changes or events on the application level, or even *implicit* events that indirectly result from some action (often not even controlled by a Web agent).
Below, we give a list of possible event types:

- system events, e.g., temporal events (local),
- updates to local data (local),
- incoming messages, including queries, answers (local),
- transactional events (commit, confirmations etc) (local),
- updates of data anywhere in the Web (remote),
- implicit "events" that occur somewhere in an application, and that are (possibly) represented in explicit data. Rule-based behavior of agents is often based on an abstract specification of application-level events (local/remote, semantic level).

Example 1.4 (Events) *The following are events:*

- *internal events like updates of the local database are atomic events.*
- *Simple local atomic events are also e.g. "it is 12:00 h" (temporal event), "user X.Y., logs on".*
- *An atomic remote event is e.g. "the book* XML Query *by Lehner et al is added to the dpunkt publisher's catalogue Web page". In this latter case, the detection can either be by notification (registered, "push"-communication), by polling ("pull"-communication), by a publish/subscribe system (where dpunkt publishes), or by submitting a suitable query to a continuous query service.*
- *The event E_1 = "the price for one liter of fuel (at a given petrol station) raises above 1.50 Euro" is also an atomic remote event, that can e.g. be used in a composite event of the form "if E_1 and then E_2 then do action". E_1 is detectable, but only indirectly after some update (in case of an update notification), or by a regular query (polling). The ECA language should support behavior rules of this kind.*
- *The event "Goal by Figo in the 85th minute" is an event which can be used in several ways: Note that the event "received a message that Figo scored a goal in the 85th minute" is another event that contains information about an event. Here also the difference between actual time (e.g., 19:55) and transaction time (e.g., 20:03) of obtaining knowledge about the event is relevant.*
- *The declarative event "today's train ICE 800 from Frankfurt to Munich is announced to be at least 20 minutes late" can be used in a reactive specification, although its detection is not clear. It can be detected e.g. via a Web page where such situations are announced, or via checking when it actually departs from a point in the way, say in Stuttgart.*
- *the implicit event "X = the German national team does not win the European Championship" is an event that can e.g. be used in a rule "if ... then ...". Detection of the event could be done either (i) if the EC is finished and another team has won, or – much earlier, using semantic knowledge – when the team dropped out of the competition after the group phase.*

An important requirement for the Semantic Web here is that event specification, content and event detection are as much declarative and application-level as possible; in the best case in "nearly natural language".

Explicit Communication: Messages. *Messages* are a special type of *atomic events* that are used for the *exchange* of all kind of explicit information between Web sites. Receipt of a message itself is an event, and messages can be used for exchanging information *about* events, and also for sending queries and answers.

In most cases, messages are sent directly to one or more receivers (notifications, queries, answers), but in some cases messages are sent "to all". Since a direct broadcast in the open community of the Web is not possible, *global* event bases can be employed, following the *publish/subscribe-paradigm*; this makes sense especially on the Semantic Web with application-specific event bases (possibly also combined with ontology-based services).

Remote Events. As illustrated above, detection of, and reaction upon, remote events is an important feature for the Semantic Web. An event detection engine must also be able to detect/discover remote events that are not explicitly communicated. This is especially the case when working with complex events (see below). It can be done by using remote event bases

(when the location of an event is known, and it is known that it is traced in an event base), or by regularly polling remote data (e.g. fuel prices at my favorite petrol station, or stock courses). In this case, again, publish/subscribe systems or continuous-query services can be applied (especially, when they maintain a history).

Implicit Events. Most of the events *can* be expressed alternatively as detection of updates of a given database (communicated via publish/subscribe systems), or by queries but, especially in the *Semantic* Web, a declarative specification from the point of view of an *event* is intended. The reduction of the detection to an actual update is then left to the semantic component.

Event Metadata. We must also distinguish between the event itself (often an application-specific event), and its metadata, like type of event (update, temporal event, receipt of message, ...), time of occurrence, the time of detection/receipt (e.g., to refuse it, when it had been received too late), the event generator (if applicable; e.g. in terms of its URI). This is in accordance with languages for communication among agents defined e.g. by FIPA[3], where besides the definition of speech acts or utterances that can be used for communication, there is the definition of a content language [FIP02d] and a message structure [FIP02a] which considers metadata such as sender, receiver, etc.

Note that all this information about time points is indispensable for synchronization on the Web within the update/event language. Event languages of active database systems usually do not make it possible to express such information because it is not needed in such systems having a centralized, common synchronization. The issue of synchronization has also been studied in the area of *agents*. An overview of studies in this area relevant for REWERSE can be found in Chapter 5.

Languages. Accordingly, we need two special sublanguages:

- a language for actual messages (XML, to be exchanged),
- a language for *describing events* (metadata, including a contents part that contains the actual event), according to an *event ontology*.

These languages should not be concerned with what this information might be, but instead make the exchange of all kind of data possible. So, an event must contain the above general information, and all other necessary information intended for its receiver is then encapsulated as data, described in an application-specific language, depending on the type of an event (e.g., in case of an update: what must be updated, and to what value(s), or the RDF fragment (person:Figo,shoot,(x:goal,when,time:'19:55'))).

Explicitly, this means that queries, answers to queries, and update requests must *not* be expressed as events, but as *contents* of the messages, expressed in their *own* languages. Messages are e.g. used for "update notification", i.e. telling that an update has been performed (either as a response to an update request, or just as an information), or for "update rejection notification", i.e. for telling that an update has been rejected. Note here the similarity with FIPA communicative acts [FIP02b], where a fixed set of acts for communication among agents are established (e.g. inform, propose, agree, refuse, accept, request, etc). However, we are not assuming here that the set of possible acts are defined a priori, nor are we aiming at dealing with all the panoply of acts proposed by FIPA for dealing with communication between intelligent agents.

[3]See `http://www.fipa.org`

Event Detection and ECA Rules

Usually, ECA rules are rule *patterns* that contain variables to be bound in the event part that are communicated to the condition and action parts (e.g., in SQL triggers in form of the `old`) variable; in most cases, also the `new` value is recorded).

Complex Events, Event Algebras

Besides atomic or primitive events, reactive rules can also use the notion of *complex events*, e.g., "when E_1 happened and then E_2 and E_3, but not E_4 after at least 10 minutes, then do A". Complex events are usually defined in terms of an *event algebra*[4]. This area has been investigated in-depth in the area of *active database systems*.

When considering event algebras for reactive rules on the Web, not only the straightforward conjunctive, disjunctive and sequential connectives, but also e.g. "negative events" in the style that "when E_1 happened, and then E_3 but not E_2 in between, then do something" should be considered. Also, "aperiodic" and "cumulative aperiodic" events should be considered.

It is desirable that event sequences can be combined with requirements on the state of resources at given intermediate timepoints, e.g. "when at timepoint t_1, a cancellation comes in and somewhere in the past, a reservation request came in at a timepoint when all seats were booked, then, the cancellation is charged with an additional fee". In this case, the event detecting engine has to state a query at the moment when a given event arrives. For being capable of describing these situations, a formalism (and system) that deals with sequences of events and queries is required such as, e.g., [Har79, BMP81, CE81, EL85, Gab89, Rei93, CKAK94, BK94, Cho95b, Sin95, SW95, MZ97, LLM98].

Work on complex events does not only define the semantics of events and complex events, but in general also describes algorithms for efficient detection and tracing of events (e.g., incremental *residuation* in [Sin95], incremental bookkeeping in [Cho95c, SW95]). They are in general restricted to the area of distributed/active databases where the location and communication of events is fixed. In the context of the Web, the *global handling* of events must also be investigated.

All investigations have also to deal with the aspect of incomplete knowledge, e.g., by weakening the ECA semantics to "if I become aware of some event, then do something" (which is the same way as if humans maintain a Web page), and to add the possibility to apply reactions later on if necessary. In this setting, updates to the event base must trigger re-evaluation of all ongoing event-checking activities. In case of cumulative events, the contents of the contributing events must be collected accordingly.

Event Bases. Models, languages and strategies for event bases (the local ones, and possibly central ones, dedicated to a given application domain) have to be defined, possibly with separate communication strategies for exchanging knowledge about events with cooperating event bases.

1.4.2 Conditions

Conditions are in fact tested by queries against the local knowledge, knowledge of remote resources, and the Web as a whole. Note that the evaluation of "condition-queries" can possibly return values that act as parameters for the responding action.

[4]For a survey on event algebras, see Section 2.7.

1.4.3 Actions

Actions, which are what in the end may cause *reactivity and evolution* of the Web, can be of several kinds: sending a message (that can also be a query - to be answered information for another node -including answering a query), an external action, or a (local) update. The action part of an ECA rule can also contain a *transaction*.

Updates

Updates are partitioned into local updates and remote updates – the latter actually take place via messages. Nevertheless, concerning reasoning about correctness, an effect such as "resource X is updated as ..." has to be used (e.g. when reasoning about remote knowledge). Communication of updates and confirmations will be discussed later in Section 1.5.

Local knowledge is defined by facts and optional derivation rules, and behavior is defined by reaction rules. All of this local knowledge is uniformly represented in XML, and is updatable, in the sense that the update language to be developed must be capable of changing facts, derivation rules and reactive rules. Here we may rely on the studies done in the context of logic programming about updating derivation and reactive rules (see Section 3.3).

Remote Method Calls

Activities of remote nodes can be invoked by sending them a message with an update statement, or, for general method calls to Web Services, the SOAP [soa00] protocol can be used on the Web. Here, also WSDL/UDDI can be employed for finding appropriate services in the Semantic Web.

External Actions

External actions are the eventual effects of the Web on its outside environment (e.g. printing a bill). For reasoning about correctness on the application level, they have to be considered explicitly as part of the information system. For example, the specification of use cases or activities in UML is in general not done on the level of internal databases ("a reservation is stored in the DB"), but on the application level ("a client booked a hotel room"). Thus, they are an important part of the specification of a system. They can also be explicitly mentioned in application-level ECA rules – as such, they are "fed back" as events (e.g. "if somebody booked a room at Sheraton, and at his flight arrives later than 22:00, a taxi must be sent").

Transactions

Transactions are another point of view of ECA rules: they can be implemented themselves by ECA rules, but they can also serve as a concept of their own that is managed by a separate *transaction manager*. Similarly to ECA rules, transactions combine events, queries, and updates. We intend to explore the use of Transaction Logic [BK94] and of Evolving Logic Programs [ABLP02] for this purpose.

Distributed, and nested transaction management, based on local transactional functionality of nodes, is required for evolution and reactivity on the Web.

1.4.4 ECA Rule Markup Languages

We need a common, global language of rules (and sublanguages for events) that covers evolutionary and reactive behavior (especially, updates) of resources. Having ECA rules and derivation rules together, each rule must contain information about its "sublanguage" and its semantics. Research in this area is done by REWERSE's *WG I1: Rule Markup*.

1.4.5 General (Re)active Resources

The behavior of Web nodes is generally not restricted to ECA rules as described above for implementing updates, but there can be "any" behavior – as e.g. for Web Services. For such services, the *Agents* paradigm is often appropriate (see Chapter 5). Communication with them takes place via common Web protocols (SOAP etc.). In case that reasoning about such a service is needed, some kind of formal specification of it is assumed.

1.5 Communication Structure and Propagation of Knowledge

We see the upcoming Semantic Web as a living organism consisting of autonomous resources and groups or communities of resources (sharing e.g. a common ontology and trust policies). Concerning communication in this Semantic Web, and during its development from the Conventional Web, several issues come into play. Concerning the actual implementation of communication provided by basic communication protocols like TCP/IP or HTTP and higher level protocols like SOAP, REWERSE will most probably use the existing concepts. The communication *structure* in the Web (for distributed query answering, updates etc.) is an important matter when realizing Web-wide behavior, as already pointed out for the Conventional Web in Section 1.3 when considering propagation of updates. The Semantic Web will use similar general communication patterns, enriched by semantics-oriented features. Establishing communication requires knowledge about capabilities and behavior of participating resources. Thus, every resource must be distinguished by its capabilities, both from the point of view as of a knowledge base, and from a directly functional point (i.e. its interface, e.g. considered as an XML database or as a Web Service). In general, for establishing and maintaining communities, metadata about available resources must be communicated. In the Conventional Web, this is done via WSDL/UDDI; for the Semantic Web such services will be lifted to the semantic level; e.g. based on OWL-S [OWL03].

Conventional Web. In the Conventional Web, without semantic metadata about resources, communication concerning query answering and *evolution* of the Web is bound to definite links between resources. In this setting, evolution takes place if a resource (or its knowledge) evolves locally, and another resource that depends upon it also evolves (as a reaction).

Example 1.5 *Consider the Website of an airline, where the flights with some information (times, prices etc.) is given, and another resource that compares the offers of several airlines and shows e.g., the cheapest connection for a set of common connections. If an airline page changes, the comparison page must also change.*

Here resources can be classified according to their role as information providers/consumers:

- resources providing information: these are data "sources" that are updated locally by their maintainers (e.g., the flight schedule database of *Lufthansa*). Other information sources may use their information – we call such resources *information sources*;
- resources that combine information from other resources, and that in course provide this information (e.g. travel agencies providing packages of flights and train connections) – we call such resources *information transformers*.
- resources that use information from others but are not (or are only in small scale) used as information providers (e.g., statistics about the average load of flights).

The above roles of resources induce different handling of the *communication paths*. One possibility is to have a *registered communication path*, where the "provider" knows about the "client" who uses its data. In another possibility, the user of the information knows his "provider", but not vice versa.

Additionally, information transformers can be classified according to the way how their knowledge is represented, and how it is maintained. An information transformer may not materialize its knowledge but, rather, answer queries only by transforming them into queries against sources/transformers via its application logic. Alternatively, an information transformer may maintain an own (materialized) database that is populated with data that is derived from information from other sources/transformers.

The application logic of non-materializing transformers is in general Datalog-like specified, where their *knowledge base* is represented via *derivation rules*. There is no internal knowledge except for the rule base. Thus, answers are always computed from the current state of the used transformers or sources. There is also no knowledge maintenance problem; however, such resources totally depend on the 24/7 availability of their information providers.

The application logic of materializing transformers can either be specified in a Datalog-style, or by *reactive rules* that state how to update their information according to incoming messages (or a combination of both). In the latter case, the resource is more independent from its providers, but the *view maintenance problem* comes up.

Using peer-to-peer-communication, the propagation of knowledge between sources, which is obviously desirable to keep resources up-to-date, can then be done in two distinct ways:

- Push: an information source/transformer informs registered users of the updates. A directed, targeted propagation of changes by the *push* strategy is only possibly along registered communication paths.
- Pull: resources that obtain information from a fixed informer can *pull* updates by either explicitly asking the informer whether he has executed some updates recently, or can regularly update themselves based on queries against their providers.

Depending on the functionality of the participating nodes, often a *push* strategy is not possible. This is for instance the case when the data source is a simple Web page or XML database, or when it refuses to individually inform all nodes that are interested. Often, a *pull* strategy is globally inefficient, since lots of clients would pose the same query continuously. Already in the Conventional Web, there are several kinds of services that provide special forms of propagation:

- *Publish-and-subscribe* services (see e.g. [TRP+04]) receive messages from publishers and notify subscribers if the messages match the subscriptions. Here, the communication follows a pure *push* pattern, i.e., information (published items) are pushed from their originators to the pub/sub system, and derived information (notification about changes) is pushed from the pub/sub service to its subscribers.

In the above example, the Lufthansa and other airlines could act as a publisher against a pub/sub service, and the comparison service could subscribe to this service (submitting the exact XQuery queries against the data published by Lufthansa etc). Note that on the Semantic Web, the subscription could be done in a high-level way, stating only "inform me about flight schedules".

- *Continuous query systems* (see e.g., NiagaraCQ [CdTW00]) allow users to "register" queries at the service that then continuously states the query against the source, and informs the user about the answer (or when the answer changes). Here, communication combines *pull* and *push*: the CQ system *pulls* information from sources, and *pushes* derived information to the end user.

  In the above example, the comparison service submits the exact XQuery to the CQ service. Other consumers submit possibly overlapping queries, and the CQ service polls the airline pages and extracts appropriate answers for each subscriber's query.

In both cases, the information consumer is correctly informed about the current state and changes. Thus, propagation of updates is possible in the Conventional Web – using a plain XML query language, which works as long as the query that is used for obtaining the relevant information has not to be syntactically changed.

This is an important aspect for update propagation on the *Semantic* Web: since the specification is then on a semantic, application level, the actual representation of the knowledge does not matter.

Semantic Web. The communication means described above for the Conventional Web can be used, extended by the fact that the propagation is less vulnerable to changes of the information source structure, since a *semantic* query can now be used. Note that it is still required to know which node actually provides the information.

But then, the advantages of the Semantic Web come up: the query can be stated, e.g., on the RDF level against "the Web" to be answered by any other node that uses an ontology that overlaps in the relevant notions. Considering communication, the meaning of "the Web" has to be resolved; depending on the needs and capabilities of nodes on the Web, different "social" communication structures are possible:

- a set (*community*) of nodes that share a common ontology and provide complementary (more or less overlapping) data. Communities can have different intra-community structure which in course has implications to the interface towards the user.

  In a centralized community, there exists a central resource that does the mapping and integration tasks, and that acts as a *portal*, serving as external interface for querying, and also for service negotiations (e.g., communicating the ontology to the outside). Participants of the community communicate with the portal, and possibly also with each other.

  In a non-centralized communities, each resource performs the mapping task by itself when it is queried, and (sub)queries are forwarded to relevant nodes in the community to be answered.

  Usually, such communities are formed intentionally by nodes that cooperate permanently (e.g., tourist offices in a region, airlines, travel agencies, car rental companies etc. – a tourist information system is one of the prospective testbed scenarios in REWERSEor research groups in bioinformatics), agree in certain consistency requirements, and that trust each other (see REWERSE's *WG I2: Policies*); there is also a well-defined flow of updates. For an application area, there can be several such communities.

Such communities provide a natural notion of community-global state, as the union of the knowledge in each community resource. For maintaining the consistency of the global state in this environment of autonomous resources, appropriate communication and reasoning mechanisms are needed; see Section 1.6.

- ad-hoc peer-to-peer communication: appropriate partners must find each other; here a distributed P2P kind of CORBA or UDDI here can be used. After finding potential partners, communication must be established (concerning negotiation of ontology overlaps, interfaces, trust etc.) (see again REWERSE's *WG I2: Policies*).

We intend to take basic communication mechanisms and Semantic Web policies as taken for building reactivity and evolution on them. In I5's starting research in evolution and reactivity for the Semantic Web, we mainly consider evolution inside a community.

For inter-community communication , the mapping between the ontologies must be done by distinguished *bridge* resources that belong to two or more communities and that *mediate* between the ontologies by mapping rules between the ontologies, and also mapping communication messages between the communities. A great amount of work exists on the definition of such mappings and mediation for querying (see e.g. [BGK+02, BS03, HIST03, RB01, Kle01, CDGL+04] and references therein).

The tasks of mapping data between, as well as the necessary (ontology) mappings required for this, though quite important, are not particular to evolution on the web, nor to reactivity. Rather, it is a general issue for querying the Semantic Web and, as such, we will rely here on the results produced by the corresponding Working Groups of REWERSE. Also, lifting the initial assumption that communities can be accessed by a common semantics level, will mainly depend on the existence of query languages and ontology mappings, to be developed elsewhere in REWERSE.

I5 will also build on the work of *WG I4, "Reasoning-Aware Querying"* for their work on evaluating intra- and inter-community queries, including their ontology formalisms and ontology mappings. These mapping can then be extended to update propagation and message transformation between different ontologies.

1.6 Reasoning on the Semantic Web

Reasoning on (and about) the Semantic Web is an area that deserves further research on various topics. Not all of these topics are directly related to evolution and reactivity and, as such, will not be directly studied within I5, nor will this document provide a systematic overview on extant related work. In this section we point out some important aspects that should be taken into account, especially those with influence on evolution.

Reasoning inside Communities. As stated above, the notion of community-global state, as the union of the knowledge in the resources of a community that shares a common ontology and agrees in certain consistency requirements has to be defined and investigated. Reasoning about consistency and consistency preservation is obviously related to I5.
Outside a community, consistency, and completeness can in general not be guaranteed or assumed.

Uncertainty and Incomplete Knowledge Throughout the Web. The Semantic Web gives the user (and its resources) an "infinite" feeling (no instance can ever claim to collect all knowledge of the Web as a whole). Thus, "global" reasoning inside the Web must be based on a

"as-far-as-I-know"-logic: all reasoning *on* the Web has to cope with uncertainty and incomplete knowledge, and, in case of derived facts in general also with nonmonotonic "belief" revisions (negation, aggregates).

Dealing with Quality of Information. Obviously, the Web (and also the Semantic Web) contains inconsistent information - both by simply erroneous or outdated information. Additionally, many information resources provide opinions, beliefs, conjectures and speculations. Often, these should also be handled (e.g., for answering questions like "what stocks in automobile industry are recommended to buy?"). Even, information *about* (here, we do not mean metadata, but e.g., evaluations of the trustworthiness and the quality, or comments about statements of other resources) resources is available. Thus, reasoning has to cope with inconsistencies, incompleteness, and with modal information.

Inconsistencies and Consistency Management. Inconsistencies can occur inside the information of a resource (to be avoided), between resources of a "community" (in most cases, also to be avoided), and across the whole Web (immanent). The latter ones cannot be avoided, but must be *handled* by each resource.

For consistency management, in addition to simply "importing" updates, resources must be able to apply additional reasoning. There may be the case that the receiver of an update does not want to implement an update because he does not believe it. Then, the community-global knowledge becomes either inconsistent (when dealing with facts), or there are different "opinions" or "presentations" (e.g., when handling news from different news services). Additionally, a resource can come to the decision that one of its informers must be replaced by another one. So, the communities can be subject to internal restructuring (still sharing the same ontology, but with different communication paths).

The Semantic Web itself also has to act in a self-cleaning way: Resources that provide erroneous or outdated information can be notified, requested for update, and potentially sanctioned by others. For doing this, instances with powerful reasoning capabilities (and equipped with some authority!) are needed. Communities inside the Semantic Web will turn out to be useful for this kind of *quality management*.

Reasoning about Dynamic Aspects. Mainly for reasoning *about* evolution and reactivity in the Web, a logic for reasoning about several states is needed. We need to be able to express evolution and reactivity of individual resources, and of the global state. Here again, two issues will be handled:

- Low number of successive states for describing immediate reactions, e.g., for update propagation and consistency maintenance; here, a detailed reasoning on the states is required, and a logic in the style of classical temporal logics, LTL or CTL with a Kripke semantics based on a given semantics for individual states, could be used.
- Activities involving a higher number of states for describing *transaction*-like activities on the Web, in most cases on a higher level of abstraction. Here, the dynamic aspect is dominating over the actual data, and a logic in the style of Transaction Logic [BK94] is useful.

Especially, one further needs to find and investigate suitable formalizations for describing the autonomous, distributed nature of the knowledge and its change in time, and to express a global semantics and reasoning principles about the *Web Evolution and Reactivity* rule language in these formalisms. The formalization of rules must be investigated from the "specification", the "implementation", and the "planning" point of view. Here,

Actions in the Head: "classical" ECA rules that are evaluated "bottom-up" as in many explicitly state-oriented approaches like e.g. *progressive rules* in Production Rules [AWH95,

PV95], Statelog [LLM98], Evolving Algebras aka Abstract State Machines [Gur88], or logic programs updates [ABLP02]. If the rule body evaluates to true in some state, the update specified in the head must be applied.

Actions in the Body: The execution of (basic) actions is triggered in the rule *body* in the top-down approach, e.g. in *Transaction Logic* [BK94]: The "target" in the rule head is implemented by the expression given in the rule body, consisting of queries (i.e., conditions that can be evaluated at any timepoints in the state sequence), subgoals (in the same way specified how to reach in rules), and elementary actions (i.e., in Transaction Logic, updates).

Example 1.6 (Actions in the Body/Actions in the Head) *For a flight reservation, the rule:*

```
reservation(Name,Flight) :- available(Flight,Place),
    insertTicket(Name,Flight,Place), printTicket(Name,Flight,Place).
```

describes how a flight reservation is accomplished by inserting a booking into the database and printing it.
On the other hand,

```
insertTicket(Name,Flight,Place), printTicket(Name,Flight,Place) :-
            reservation_request(Name,Flight), available(Flight,Place).
```

expresses the same in a production-rule-like syntax where the actions to be taken are given in the head of the rule.

The "target-driven" top-down/actions-in-the-body approach is mainly useful for planning how to reach a given goal, including reasoning about complex (trans)actions. The "event- and data-driven" bottom-up approach directly defines actual "reactive" behavior, where the question of how a given target can be reached requires additional reasoning *about* the state sequence generated by the execution of rules.

Summary and Outlook

The view described up to this point is to result in a rule-based infrastructure for implementing, and reasoning about, evolution and reactivity on the Web.

In it, we start from the non-semantic level where resources can mainly be divided into data-oriented resources that are usually described by an XML schema, and service-oriented resources (e.g. Web services), described by some interface description mechanism (e.g. OWL-S or WSDL). An intermediate ECA rule language for the Web will be a first step.

In the following chapters, we give a more detailed account on the state of the art in some relevant areas.

Chapter 2

Logics for Updates and State Change

Kripke structures serve as a *generic* model-theoretic framework for multi-state structures: the semantics of the individual states is given by some single-state interpretations, and the Kripke structure provides the "infrastructure" that connects the states. Some (arbitrary) logic is used for the single-state interpretations, and this logic is extended, in a modular way, with additional concepts for handling the multi-state aspects. This can be done by modalities (in our situation, temporal modalities, but modalities of knowledge and belief are also often used).

Many approaches to multi-state reasoning use Kripke structures explicitly; here, temporal logics will be described. Other, often specialized formalisms, extend single-state formalisms with a notion of state (in which Kripke Structures are -more or less explicitly- the model of choice).

Yet other formalisms -although basically mappable to Kripke Semantics- put emphasis on the dynamic aspects, whereas the single states and their properties become less important (Transaction Logic, and, even much stronger, process calculi).

2.1 Kripke Structures

Assume some logic (e.g., first-order logic) to describe individual states. A (first-order) *Kripke structure* is a triple $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$ where $\mathcal{G}$ is a set of states (to be interpreted as states or possible worlds), $\mathcal{R} \subseteq \mathcal{G} \times \mathcal{G}$ is an *accessibility relation*, and M is a function which maps every state $g \in \mathcal{G}$ to a (first-order) structure $\mathcal{M}(g) = (M(g), \mathcal{U}(g))$ over Σ with universe $\mathcal{U}(g)$. $\mathcal{G}$ and $\mathcal{R}$ are called the *frame* of $\mathcal{K}$. A *path* p in a Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$ is a sequence $p = (g_0, g_1, g_2, \ldots)$, $g_i \in \mathcal{G}$ with $\mathcal{R}(g_i, g_{i+1})$ holding for all i.

2.2 Modal Temporal Logics – based on Kripke Structures

As mentioned above, Kripke structures provide just a multi-state "infrastructure": a suitable single-state-logic must then be chosen for an application, which is then extended to Kripke structures. Here, *temporal extensions*, where the Kripke structure is interpreted as a temporal structure, are suitable for our project (note that a large research area deals with "possible

worlds semantics" – they use the same basic modalities as below, but with other intentions, and another semantics of the accessibility relation $\mathcal{R}$ of the Kripke structure). Even in the area of temporal applications, there are different interpretations of Kripke structures and temporal modalities.

Due to the historical development of modal logics, the modal operators $\Box$ and $\Diamond$ were introduced. $\Box F$ stands for F is necessary true, resp. F holds in all possible worlds, and $\Diamond F$ for F is possibly true, resp. there can be some world where F holds. Translated to modal logic of time (temporal logic), the operators are interpreted as: $\Box F$ – "always" (F holds in all subsequent states), and $\Diamond F$ – "sometimes" (F eventually holds).

For reasoning in temporal Kripke structures, there are two alternatives: *linear time* considers a single path, whereas *branching time* considers the whole tree-like structure.

2.2.1 Linear Time Temporal Logics

The most intuitive idea for interpreting temporal logic is a sequence of states. Here, the basic operators of temporal modal logic are others, having a pure temporal semantics: $\circ$ ("nexttime") and until:

- $\circ F$: in the next state, F holds.
- F until G: there is a subsequent state where G holds, and in all states between now and this state, F holds.

The semantics of the temporal modal operators $\Diamond$ and $\Box$, is equivalently defined via until (note that there is a also an inductive definition based on $\circ$ which is typically used for model checking-like approaches):

- $\Diamond P := \mathsf{true}\ \mathsf{until}\ P$ and
- $\Box P := \neg\Diamond\neg P$.

The Logics PLTL and FOTL. Linear Temporal Logic LTL (here described, as propositional PLTL; analogously there is first-order FOTL) extends propositional logic with the above temporal operators: each state is a propositional interpretation, and the states are connected as a *linear* Kripke structure (or, a single path in a branching Kripke structure is considered).

The language of LTL formulas is defined as follows:

- Every (propositional or first-order) formula is an LTL formula.
- With F and G LTL formulas, $\circ F$, $\Box F$, $\Diamond F$ and (F until G) are LTL formulas.

The satisfaction relation $\models_{\mathrm{PLTL}}$ (for short also denoted by $\models$) is defined according to the inductive definition of the syntax with respect to a propositional (infinite) linear Kripke structure $\mathcal{K} = (\mathcal{G} = \{g_1, g_2, \ldots\}, \{(n, n{+}1) \mid n \in \mathbb{N}\}, \mathcal{M})$, based on the propositional satisfaction relation $\models_{FOL}$:

Let $g = g_i$ a state in $\mathcal{K}$, A an atomic formula, F and G LTL formulas and χ a variable assignment. Then,

$$
\begin{array}{lcl}
(g,\chi) \models A & :\Leftrightarrow & (M(g),\chi) \models_{PL} A \ , \\
(g,\chi) \models \neg F & :\Leftrightarrow & \text{not } (g,\chi) \models F \ , \\
(g,\chi) \models F \wedge G & :\Leftrightarrow & (g,\chi) \models F \text{ and } (g,\chi) \models G \ , \\
(g_i,\chi) \models \circ F & :\Leftrightarrow & (g_{i+1},\chi) \models F \ , \\
(g_i,\chi) \models F \ \mathsf{until} \ G & :\Leftrightarrow & \text{there is a } j \geq i \text{ s.t. } (g_j,\chi) \models G \\
 & & \text{and for all } k : i \leq k < j,\ (g_k,\chi) \models F \ .
\end{array}
$$

2.2.2 Branching Time Temporal Logics

Applying the classical temporal modalities $\Diamond$ and $\Box$ (without $\circ$ and until) in a *branching* structure leads to surprising interpretations: Whereas in linear time logic, $g \models \Diamond F$ means that F will eventually hold in the possible future ("sometimes"), the same formula for branching time means that there *is a future*, where F will eventually hold ("not never"). For this aspect and the (dis)advantages of linear vs. branching time logic, see [Lam80] (L. Lamport: "'Sometimes' is Sometimes 'Not Never'") and [EL85] (E. A. Emerson and C.-L. Lei: "Modalities for Model Checking: Branching Time Strikes Back") and several other papers.

The Logic UB. For combining the expressiveness of both linear and branching time logic, in [BMP81] the logic UB (*unified branching time*) has been introduced. UB solves the problem by *path quantifiers* A ("on all paths") and E ("there exists a path such that ..."). With this, the logic is fundamentally different from the historical modal logics. The modalities of UB are the following: $\mathsf{A}\Box$, $\mathsf{E}\Box$, $\mathsf{A}\Diamond$, $\mathsf{E}\Diamond$, $\mathsf{A}\circ$ and $\mathsf{E}\circ$.

The Logic CTL. CTL (*Computation Tree Logic*) has been introduced in [CE81], originally as propositional logic for branching time (by extending propositional CTL with first-order atomic formulas and quantifiers, first-order CTL is obtained, which is described below). It is based on UB and additionally provides the binary operator until. In the syntactical definition, the modalities do not consist of the combination of a path quantifier and a modal operator, but these features are distinguished:

- temporal modal operators $\circ$, $\Diamond$, $\Box$ and until (although $\Diamond$ and $\Box$ can be expressed by until, they are used here as "basic" operators to obtain the syntax definition described below),
- an existential path quantifier E and a universal path quantifier A.

In CTL, two different types of formulas can be distinguished: State formulas that hold *in* a state (all first-order formulas are state-formulas), and path formulas, that hold on paths, i.e., on *sequences* of states.

The language of CTL-formulas does not allow arbitrary combinations, but is defined as follows:

- Every first-order formula is a CTL-state formula.
- With F and G CTL-state formulas, $\neg F$, $F \wedge G$ and $F \vee G$ are CTL-state formulas.
- With F a CTL-state formula and x a variable, $\forall x : F$ and $\exists x : F$ are CTL-state formulas.
- With F and G CTL-state formulas, $\circ F$, $\Box F$, $\Diamond F$ and $(F \ \mathsf{until} \ G)$ are CTL-path formulas.
- With P a CTL-path formula, $\neg P$ is a CTL-path formula.
- With P a CTL-path formula, $\mathsf{A}P$ and $\mathsf{E}P$ are CTL-state formulas.
- Every CTL-state formula is a CTL-formula.

With the above definition, in CTL every (possibly negated) modal operator is immediately preceded by a path quantifier.

The satisfaction relation $\models_{\mathrm{CTL}}$ (for short also denoted by $\models$) is defined according to the inductive definition of the syntax with respect to a first-order Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$, based on the first-order satisfaction relation $\models_{PL}$:

Let $g \in \mathcal{G}$ be a state, $p = (g_0, g_1, \ldots)$ a path in $\mathcal{K}$, A an atomic first-order formula, F and G CTL-state formulas, P a CTL-path formula and χ a variable assignment. Then,

$$
\begin{array}{lll}
(g,\chi) \models A & :\Leftrightarrow & (M(g),\chi) \models_{PL} A \ , \\
(g,\chi) \models \neg F & :\Leftrightarrow & \text{not } (g,\chi) \models F \ , \\
(g,\chi) \models F \wedge G & :\Leftrightarrow & (g,\chi) \models F \text{ and } (g,\chi) \models G \ , \\
(p,\chi) \models \circ F & :\Leftrightarrow & (g_1,\chi) \models F \ , \\
(p,\chi) \models F \ \mathsf{until}\ G & :\Leftrightarrow & \text{there is an } i \geq 0 \text{ s.t. } (g_i,\chi) \models G \text{ and} \\
 & & \text{for all } j : 0 \leq j < i,\ (g_i,\chi) \models F \ , \\
(p,\chi) \models \neg P & :\Leftrightarrow & \text{not } (p,\chi) \models P \ , \\
(g,\chi) \models \mathsf{E}P & :\Leftrightarrow & \text{there is a path } p = (g = g_0, g_1, \ldots) \text{ in } \mathcal{K} \text{ s.t. } (p,\chi) \models P \ .
\end{array}
$$

The semantics of the modal operators $\Diamond$, $\Box$, and of the path quantifier A is defined via until and E:

$$\Diamond P := \mathsf{true}\ \mathsf{until}\ P \quad , \quad \Box P := \neg\Diamond\neg P \quad , \quad \mathsf{A}P := \neg\mathsf{E}\neg P \quad .$$

Thus, the semantics of the derived operators is formally given as

$$
\begin{array}{lll}
(g,\chi) \models F \vee G & \Leftrightarrow & (g,\chi) \models F \text{ or } (g,\chi) \models G \ , \\
(p,\chi) \models \Box F & \Leftrightarrow & \text{for all } i,\ (g_i,\chi) \models F \ , \\
(p,\chi) \models \Diamond F & :\Leftrightarrow & \text{there is an } i \text{ s.t. } (g_i,\chi) \models F \ , \\
(g,\chi) \models \mathsf{A}P & \Leftrightarrow & \text{for all paths } p = (g = g_0, g_1, \ldots) \text{ in } \mathcal{K}\ (p,\chi) \models P \ , \\
(g,\chi) \models \forall x : F & \Leftrightarrow & \text{for all } d \in \mathcal{U}(g),\ (g, \chi_x^d) \models F \ .
\end{array}
$$

The Logic CTL^+**.** In CTL, some composite modal operators cannot be expressed:

$$
\begin{array}{lcl}
F \ \mathsf{unless}\ G & := & (F \ \mathsf{until}\ G) \vee (\Box(F \wedge \neg G)) \\
F \ \mathsf{before}\ G & := & \neg G \ \mathsf{until}\ (F \wedge \neg G) \\
F \ \mathsf{atnext}\ G & := & (\neg G \ \mathsf{until}\ (F \wedge G)) \vee \Box\neg G
\end{array}
$$

Thus, the syntax is extended to CTL^+ [EH83] by allowing composition of path formulas by the usual logical connectives:

- With P and Q CTL^+-path formulas, $P \wedge Q$ and $P \vee Q$ are CTL^+-path formulas.

The semantics is defined straightforwardly: Let $p = (g_0, g_1, \ldots)$ be a path in $\mathcal{K}$, and P and Q CTL^+-path formulas.

$$
\begin{array}{lll}
(p,\chi) \models P \wedge Q & :\Leftrightarrow & (p,\chi) \models P \text{ and } (p,\chi) \models Q. \\
(p,\chi) \models P \vee Q & :\Leftrightarrow & (p,\chi) \models P \text{ or } (p,\chi) \models Q.
\end{array}
$$

As a derived modal operator, unless, definable in CTL^+as $F \ \mathsf{unless}\ G \equiv (F \ \mathsf{until}\ G) \vee \Box(F \wedge \neg G)$, plays an important role in reasoning about processes.
In [EH82] and [EH83] it has been shown that CTL and CTL^+ have the same expressibility, and an algorithm for transforming CTL^+-formulas to CTL-formulas is given.

The Logic CTL*. The most expressive logic of the CTL family is CTL* [EH83] which allows nested modal operators and additional combinations of first-order quantifiers and path quantifiers:

- Every first-order formula is a CTL*-path formula.
- With P and Q CTL*-path formulas, $\circ P$, $\Box P$, $\Diamond P$ and (P until Q) are CTL*-path formulas.
- With P a CTL*-path formula and x a variable, $\forall x : P$ and $\exists x : P$ are CTL*-path formulas.

The semantics of the additional formulas (including derived modalities) is also defined straightforwardly: Let $p = (g_0, g_1, \ldots)$ be a path in $\mathcal{K}$, where $p|_i$ denotes the suffix $p|_i = (g_i, g_{i+1}, \ldots)$ of p beginning with g_i, which is also a path in $\mathcal{K}$. Let F a CTL*-state formula, P and Q CTL*-path formulas and χ a variable assignment.

$$
\begin{array}{lll}
(p,\chi) \models F & :\Leftrightarrow & (p \downarrow_0, \chi) \models F \quad \Leftrightarrow \quad (g_0, \chi) \models F. \\
(p,\chi) \models \circ P & :\Leftrightarrow & (p|_1, \chi) \models P. \\
(p,\chi) \models \Box P & :\Leftrightarrow & \text{for all } i : (p|_i, \chi) \models P. \\
(p,\chi) \models \Diamond P & :\Leftrightarrow & \text{there is an } i \text{ s.t. } (p|_i, \chi) \models P. \\
(p,\chi) \models P \text{ until } Q & :\Leftrightarrow & \text{there is an } i \geq 0 \text{ s.t. } (p|_i, \chi) \models Q \text{ and} \\
 & & \text{for all } j : 0 \leq j < i,\ (p|_j, \chi) \models P. \\
(p,\chi) \models \forall x : P & :\Leftrightarrow & \text{for all } d \in \mathcal{U}(\mathcal{K}),\ (p, \chi^d_x) \models P. \\
(p,\chi) \models \exists x : P & :\Leftrightarrow & \text{there is a } d \in \mathcal{U}(\mathcal{K}) \text{ s.t. } (p, \chi^d_x) \models P.
\end{array}
$$

Thus, a first-order formula also can be read as a path formula: it describes a property of the first state on a path (note that later, Transaction Logic will follow this principle).

Comparison between CTL, PLTL, and CTL*. Overviews of the above logics are given e.g. in [Eme90, Sti95]. In [MP91], the formula classes *Safety*, *Guarantee*, and *Obligation* (expressible in CTL for branching structures), and *Response*, *Persistence*, and *Reactivity* (need CTL* – note that *Fairness* is a *Reactivity*-formula, thus not expressible in CTL) are analyzed that are important for reasoning about processes.

The different expressiveness of the above logics is mirrored in the complexity for verification: the fundamental difference between CTL and PLTL/CTL* is that in the latter there can be iterated modal operators, leading to a higher complexity. The complexities for (propositional) model checking in these logics are as follows [McM93] let f be the number of operators in a given formula, V the number of states, and E the number of transitions to be checked:

CTL:	$O(f \cdot (V+E))$,
CTL with n fairness constraints:	$O(f \cdot (V+E) \cdot n^2)$,
PLTL with n fairness constraints:	$O(2^f \cdot (V+E) \cdot n^2)$ (PSPACE-complete) ,
CTL* with n fairness constraints:	same as PLTL .

Note that there are still some formulas that cannot be expressed in CTL*.

Above, temporal logics have been used for describing "simple" states in a Kripke structure. The semantics of the accessibility relation wrt. states and state change (i.e., execution of actions) is considered in e.g., in *Dynamic Logic Hennessy-Milner-Logic* (see subsequent sections), or several labelled modal logics. Furthermore, CTL is applied in the context of updating knowledge bases in EKBL (*Evolving Knowledge Base Logic*) in [EFST02b].

2.2.3 Past Tense Logic

Past Tense Logics add past-time temporal operators: $\bullet$ (previous state), $\blacklozenge$ (sometimes in the past), $\blacksquare$ (always in the past), and since (e.g., A since B), symmetrical to the future tense operators. Note that for past-tense operators there is no difference between linear and branching structures – looking backwards from a given state, there is only a single path (i.e., a single "past").

In [GPSS80], it is shown that in the propositional case, past-tense connectives do not increase the expressiveness of temporal logic. The use of modal temporal logic for executable process specifications is described in [Gab89], quite similarly to Transaction Logic (see Section 2.5).

Past tense modalities have been employed for checking temporal constraints and temporal conditions in ECA-style rules, e.g. in [Cho95c, SW95]. [Cho95c] uses full first-order past temporal logic, with $\exists$ and $\forall$ quantifiers. [SW95] replaces the quantifiers by a functional assignment $[X \leftarrow t]\varphi(X)$ that binds a variable X to the value of a term t in a given state. This ensures safety of formulas, but the full expressiveness of using a universal quantifier is not provided. Both approaches use incremental algorithms for detecting complex temporal events.

2.2.4 Dynamic Logic

Dynamic Logic [Har79, Har84, Pra76] provides a similar theoretical framework. Every language of Dynamic Logic consists of a first-order signature Σ, several logical and non-logical symbols, and a set $\mathbf{A}$ of *atomic programs*.

Dynamic Logic makes no distinction between states and the corresponding first-order structures: Γ denotes the collection of possible states, where each state $\mathcal{I} = (I, \mathcal{U}) \in \Gamma$ *is* a first-order structure over Σ. Thus, semantically, states are not explicitly seen as semantical entities in Dynamic Logic. The modal operators are labelled with *programs* given by the algebra $\langle \mathbf{A}, \{;, \cup, ^\star\} \rangle$, (";" denotes sequential composition, $\cup$ denotes alternative composition ("choice"), and * denotes iteration). With every program α, a binary relation $\mathrm{m}(\alpha) \subset \Gamma \times \Gamma$ is assigned. The main difference between CTL and Dynamic Logic lies in the scope of the modalities: The only basic modality in Dynamic Logic is $\Diamond_{DL}$ with the semantics

$$\mathcal{I} \models \langle\alpha\rangle_{DL} F \;\Leftrightarrow\; \text{there is a state } \mathcal{J} \text{ such that } (\mathcal{I}, \mathcal{J}) \in \mathrm{m}(\alpha) \text{ and } \mathcal{J} \models F$$

In agreement with the tradition, $\Box_{DL} F := \neg \Diamond_{DL} \neg F$ is defined to be the dual of $\Diamond_{DL}$. Thus, the modalities of Dynamic Logic look only one transition ahead. Thus, the *eventually*, *always*, and *until*-operators can not be expressed in DL without resorting to a fixpoint logic.

2.2.5 Hennessy-Milner Logic

In [Sti89] and [MPW92], Hennessy-Milner-Logic, $\mathcal{HML}$, a modal logic interpretation of the CCS calculus (see Section 2.6) is given. There, the modal operators $\Diamond$ and $\Box$ are interpreted in their historical sense as "possibly" and "necessarily" as in Dynamic Logic. To avoid confusion with the temporal modalities, for this interpretation the symbols $\langle i \rangle_{\mathcal{HML}}$ and $[i]_{\mathcal{HML}}$ for $i \in \mathsf{Act}^*$ are used:
The set of formulas of Hennessy-Milner-Logic, $\mathsf{Fml}_{\mathcal{HML}}$, is defined inductively as

- $\mathsf{T} \in \mathsf{Fml}_{\mathcal{HML}}$,
- $F \in \mathsf{Fml}_{\mathcal{HML}} \;\Rightarrow\; \neg F \in \mathsf{Fml}_{\mathcal{HML}}$,

- $F, G \in \mathsf{Fml}_{\mathcal{HML}} \;\Rightarrow\; F \wedge G \in \mathsf{Fml}_{\mathcal{HML}}$,
- $F \in \mathsf{Fml}_{\mathcal{HML}}$ and $a \in \mathsf{Act}^* \;\Rightarrow\; \langle\!\langle a \rangle\!\rangle_{\mathcal{HML}} F \in \mathsf{Fml}_{\mathcal{HML}}$.

(instead of $\langle\!\langle a \rangle\!\rangle_{\mathcal{HML}}$, also $\langle a \rangle_{\mathcal{HML}}$ can be written).
The satisfaction relation $\models := \models_{\mathcal{HML}}$ between processes and $\mathcal{HML}$-formulas is defined by

1. $P \models \mathsf{T}$ for all processes P ,
2. $P \models \neg F \;:\Leftrightarrow$ not $P \models F$,
3. $P \models F \wedge G \;:\Leftrightarrow\; P \models F$ and $P \models G$,
4. $P \models \langle\!\langle a \rangle\!\rangle_{\mathcal{HML}} F \;:\Leftrightarrow$ there is a process P' s.t. $P \xrightarrow{a} P'$ and $P' \models F$.

Additionally, derived expressions in $\mathcal{HML}$ are defined:

- $\mathsf{F} \equiv \neg\mathsf{T}$,
- $F \vee G \equiv \neg(\neg F \wedge \neg G)$,
- $\langle\!\langle a \rangle\!\rangle_{\mathcal{HML}} F \equiv \langle a_1 \rangle_{\mathcal{HML}} \ldots \langle a_n \rangle_{\mathcal{HML}} F$ for $a = a_1.\cdots.a_n$,
- $[\![a]\!]_{\mathcal{HML}} F \equiv \neg\langle\!\langle a \rangle\!\rangle_{\mathcal{HML}} \neg F$.

The following examples illustrate the modalities of Hennessy-Milner-Logic:

- $P \models \langle\!\langle c \rangle\!\rangle_{\mathcal{HML}} \langle\!\langle d \rangle\!\rangle_{\mathcal{HML}} \mathsf{T}$ means, that the process P can execute c, followed by d.
- $P \models [\![a]\!]_{\mathcal{HML}} \mathsf{F}$ means that P cannot execute a.
- A storage bit with read/write actions is in a state where its stored value is 0 iff $s \models \langle \mathsf{read}\ 0 \rangle_{\mathcal{HML}} \mathsf{T}$.
  Thus, the semantics of queries can also be expressed in $\mathcal{HML}$.

Here, the difference between the interpretation of the modal operators between CTL and $\mathcal{HML}$ becomes visible: In CTL, the modal operators reach into the future along a *single* path and the path quantifiers range orthogonally over all possible futures, speaking about paths not about states. In $\mathcal{HML}$, the modal operators look ahead one step on every path. Thus, it is not possible in $\mathcal{HML}$ to express properties like "P will eventually execute a" or "in the next step, P will execute a".

2.3 State-Oriented Datalog Extensions

Several approaches are based on extending "usual" logic frameworks with a notion of state. By extending Datalog with a notion of state, (re)active production rules and deductive rules can be handled in a unified way, thereby combining the advantages of active and deductive rules. Two such closely related Datalog extensions are *XY-Datalog* [Zan94, MZ97] and *Statelog* [LLM98] (see [KLS92] for an early precursor of the latter). XY-Datalog and Statelog are themselves closely related to Datalog1S [Cho95a], a query language for temporal databases.

In Statelog [LLM98], access to different database states is accomplished via *state terms* of the form $[S{+}k]$, where $S{+}k$ denotes the k-fold application of the unary function symbol "+1" to the *state variable* S. Since the database evolves over a linear state space, S may only be bound to some $n \in \mathbb{N}_0$.

A *Statelog database* $\mathcal{D}[k]$ at state $k \in \mathbb{N}_0$ is a finite set of facts of the form $[k]\, p(x_1, \ldots, x_n)$ where p is an n-ary relation symbol and x_i are constants from the underlying domain. A *Statelog rule* r is an expression of the form

$$[S{+}k_0]\, H \;\leftarrow [S{+}k_1]\, B_1,\;\; \ldots \;, [S{+}k_n]\, B_n$$

where the head H is a Datalog atom, B_i are Datalog literals, and $k_i \in \mathbb{N}_0$.

In most cases, Statelog rules are required to be *progressive*, since the current state cannot be defined in terms of future states, nor should it be possible to change past states: A rule r is called *progressive*, if $k_0 \geq k_i$ for all $i = 1, \ldots, n$. If $k_0 = k_i$ for all $i = 1, \ldots, n$, then r is called *local* and corresponds to the usual query rules. On the other hand, if $k_0 = 1$ and $k_i = 0$ for all $i \geq 1$, r is called *1-progressive* and denotes a *transition rule*. A *Statelog program* is a finite set of progressive Statelog rules.

Every Statelog program may be conceived as a standard logic program by viewing the Statelog atom $[S{+}k]\, p(t_1, \ldots, t_n)$ as syntactic sugar for $p(S{+}k, t_1, \ldots, t_n)$. This way, notions (e.g., *local stratification*) and declarative semantics (e.g., *perfect model*) developed for deductive rules can be applied directly to Statelog. A Kripke-style semantics for Statelog, based on a temporal structure of Datalog states has also been presented.

In [Zan94], XY-Datalog has been proposed as a unified framework for active and deductive rules. XY-Datalog rules can be regarded as a certain class of Statelog rules. Roughly speaking, *X-rules* correspond to nonrecursive local rules of Statelog, while *Y-rules* correspond to 1-progressive rules.

A similar extension of F-Logic [KLW95], which allows for a comprehensive treatment of state-changes and updates in databases by providing the required semantic and syntactic flexibility, has been proposed in [MSL97]. Providing as well a model-theoretic, declarative semantics as an operational semantics which is implemented by the FLORID system, State-F-Logic acts at the same time as specification language, implementation language, and metalanguage for proving properties of a system.

Relational Transducers

Whereas the semantics of the above approaches is usually defined in terms of Kripke structures, another point of view is taken in recent work on *relational transducers*. Relational transducers are specified by a state transition function, and, additionally an output function. In [AVFY98], a rule-based approach is investigated. Similar to the approach of the Situation Calculus (see Section 2.8), predicates are partitioned into fixed ones (the database relations), and state-dependent ones (input, state, output, and log). The underlying semantics is similar to the *inflationary* semantics of Datalog: the state relations accumulate all received inputs. In contrast to Statelog and XY-Datalog described above, their rules do not contain states explicitly, but always the head refers to the subsequent state.

Their investigations focus on proving correctness properties, i.e., log checking (whether a certain log is valid wrt. a given set of business rules), goal reachability and progress, temporal assertions, and transducer equivalence. For the latter, they apply the notions of bisimulation as defined originally in the area of process algebras (cf. Section 2.6).

In [Spi00], verification and its complexity issues are investigated by applying *Abstract State Machines* (see Section 2.4) to relational transducers by restricting the form of the rules.

2.4 Abstract State Machines

The "functional, algebraic" counterpart to Datalog are *Abstract State Machines*, formerly known as *Evolving Algebras*. The concept of "Evolving Algebras", has been introduced for specifying the operational semantics of processes in [Gur88, Gur95]. Evolving Algebras have originally not been introduced from the logical point of view, but for describing the operational semantics

of processes in the sense of Turing's Thesis: "Every algorithm can be described by a suitable Evolving Algebra". Thus, for any given algorithm, on any level of abstraction an Evolving Algebra can be given.

In universal algebra, a first-order structure over a signature where the equality symbol is the only relation symbol, is called an *algebra*. For emphasizing the difference to an *Evolving Algebra*, we use the term *static algebra* (which corresponds to a single state, described by functions only). For describing static algebras, similar notions are used as for first-order structures:

The *signature* Σ of a static algebra is a finite set of function symbols, each of them with a fixed arity, including 0-ary constants. The notions of terms are defined as in first-order logic.

A *static algebra* (A, S) over a (functional) signature Σ is then an interpretation of Σ, inducing an evaluation of terms. Note that every relation can be represented by its characteristic function.

An *Evolving Algebra* EA is based on static algebras, which are also called *states*. The symbols from Σ can be interpreted state-independently, thus static and dynamic functions can be distinguished (the latter also known as *fluents* in other frameworks; e.g., Situation Calculus [Rei93], GOLOG [LRL$^+$97], also [San94]).

An Evolving Algebra is given by an initial state $Z(EA)$ (which also determines the interpretation of the state-independent function symbols for all states) and a program $P(EA)$ (a set of transition rules and rule schemata) describing the change of the interpretation of state-dependent function symbols in a Pascal-like syntax.

An *elementary update rule* is an update of the interpretation of a function symbol at one location: $f(t_1, \ldots, t_n) := t_0$, where f is an n-ary function symbol and t_i are terms.

The set of rules is defined by structural induction by blocking, and conditionals (if-then); also rule schemata that contain free variables are allowed. A program $P(EA)$ of an Evolving Algebra EA is a finite set of rules and rule schemata. A program is then executed by applying rules. Note that, in contrast to Logic Programs, there is no fixpoint requirement.

For reasoning about an Evolving Algebra, a Kripke structure with appropriate states, and a suitable temporal logic can be used.

2.5 Path Structures and their Logics

For dealing with processes or transitions, several logics use *path structures*. The idea goes back to propositional *Dynamic Logic*, and the term *path model* came up with *Process Logic* [HKP82]. In contrast to the temporal logics described above, their focus is less on the states than on state transitions and paths. This idea has then been continued by *Transaction Logic* [BK94].

Assume a language L with a set AF of primitive formulas and a set AP of primitive programs (elementary actions). A (propositional) *path structure* for L is a triple (W, π, m) where W is a set of states or worlds, $\pi : AF \to 2^W$ gives the interpretation of atomic formulas, and $m : AP \to 2^{W \times W}$ provides the interpretation of atomic programs.

In contrast to Dynamic Logic, where all formulas are state formulas, the syntax of logics for path structures focusses on the notion of path formulas: The $\models$-relation relates paths to formulas. Nevertheless, in those logics, paths consisting of exactly one state actually take the role of states.

Transaction Logic. Transaction Logic $\mathcal{TR}$ [BK94] is based on path structures, reminiscent of Process Logic [HKP82]. In $\mathcal{TR}$, in contrast to modal logic where states are given as first-order structures, states are given as abstract *theories*. With this general concept, non-standard semantics, e.g., Clark's Completion [Cla78], perfect models [Prz88], or well-founded semantics

[GRS91], can be handled, allowing to characterize indefinite knowledge, deductive databases, logic programs etc. The evaluation of formulas wrt. states is provided by a *state data oracle* $\mathcal{O}^d$: for every *state identifier* i, $\mathcal{O}^d(i)$ is a set of e.g., first-order, formulas that are considered to be all and the only truths about the database state i. Transitions are given by the *state transition oracle* $\mathcal{O}^t$ which maps pairs of database states to sets of ground formulas (i.e., elementary transitions). Thus, with S denoting the set of state identifiers, $(S, \mathcal{O}^d, \mathcal{O}^t)$ gives the same information as a path model for Process Logic.

A language $\mathcal{L}$ in Transaction Logic is defined by a state data oracle $\mathcal{O}^d$ which determines the alphabet $\mathcal{P}$ of predicate symbols and the set S of identifiers of states, and a state transition oracle $\mathcal{O}^t$. $\mathcal{L}$ contains also a distinguished set $\mathcal{F}$ of function symbols which are interpreted state-independently.

The semantics is based on a version of *path structures*, i.e., the satisfaction of formulas is defined on paths, not on states: A *path* of length $k \geq 1$ over $\mathcal{L}$ is a finite sequence $\pi = \langle \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k \rangle$ of state identifiers; $\pi_1 \circ \pi_2 = \langle \mathcal{D}_1, \ldots, \mathcal{D}_i \rangle \circ \langle \mathcal{D}_i, \ldots, \mathcal{D}_k \rangle$ is a *split* of π. A path structure $\mathcal{M}$ over $\mathcal{L}$ is a triple $\langle \mathcal{U}, \mathcal{I}_\mathcal{F}, \mathcal{I}_{path} \rangle$ where

- $\mathcal{U}$ is the *domain* of $\mathcal{M}$,
- $\mathcal{I}_\mathcal{F}$ is an interpretation of the function symbols. Note that these functions are interpreted state-independently.
- $\mathcal{I}_{path}$ assigns to every path $\pi = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$ a semantic structure $\langle \mathcal{U}, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P} \rangle$ where $\mathcal{I}_\mathcal{P}$ is an interpretation of predicate symbols.

$\mathcal{I}_{path}$ is subject to two restrictions:

- Compliance with the data oracle: $\mathcal{I}_{path}(\langle \mathcal{D} \rangle) \models \phi$ for every $\phi \in \mathcal{O}^d(\mathcal{D})$.
- Compliance with the transition oracle: $\mathcal{I}_{path}(\langle \mathcal{D}_1, \mathcal{D}_2 \rangle) \models a$ whenever $a \in \mathcal{O}^t(\mathcal{D}_1, \mathcal{D}_2)$.

Atomic formulas in Transaction Logic are denoted by predicates $p(t_1, \ldots, t_n)$ (for atomic formulas, the data oracle or the transition oracle is queried). Transaction formulas are built by the connectives $\neg, \vee, \wedge, \oplus, \otimes$, and the quantifiers $\exists$ and $\forall$. Let $\mathcal{M} = \langle U, \mathcal{I}_\mathcal{F}, \mathcal{I}_{path} \rangle$ be a path structure, π a path and β a variable assignment. Then,

$$
\begin{array}{lcl}
(\mathcal{M}, \pi, \beta) \models_{\mathcal{TR}} p(t_1, \ldots, t_n) & \Leftrightarrow & (\mathcal{I}_{path}(\pi), \beta) \models_{(\mathcal{O}^d, \mathcal{O}^t)} p(t_1, \ldots, t_n) \ , \\
(\mathcal{M}, \pi, \beta) \models_{\mathcal{TR}} \phi \otimes \psi & \Leftrightarrow & (\mathcal{M}, \pi_1, \beta) \models_{\mathcal{TR}} \phi \text{ and } (\mathcal{M}, \pi_2, \beta) \models_{\mathcal{TR}} \psi \\
 & & \text{for some split } \pi = \pi_1 \circ \pi_2 \text{ of } \pi \ , \text{ and} \\
(\mathcal{M}, \pi, \beta) \models_{\mathcal{TR}} \phi \oplus \psi & \Leftrightarrow & (\mathcal{M}, \pi_1, \beta) \models_{\mathcal{TR}} \phi \text{ or } (\mathcal{M}, \pi_2, \beta) \models_{\mathcal{TR}} \psi \\
 & & \text{for } \textit{every} \text{ split } \pi = \pi_1 \circ \pi_2 \text{ of } \pi \ .
\end{array}
$$

Negation, disjunction, conjunction, universal and existential quantification are defined as usual.

Since in Transaction Logic, the internal representation of states is not predetermined, structures of any type can be used. For example, pure functional signature (e.g. static algebras), pure relational signature (Datalog), first-order, or even object-oriented (F-Logic), can be used.

Due to the restriction of $\mathcal{O}^t$ to elementary actions, parallel composition of actions in a single transition is not possible. In [BK95], an interleaving semantics for parallelism is given.

The "architecture" of a $\mathcal{TR}$ model $(\mathcal{O}^d, \mathcal{O}^t)$ is in some sense equivalent with a Kripke structure with appropriate (temporal) logics: the modelling of states corresponds to the *data oracle*, the modelling of transitions corresponds to the *transition oracle*, and the logic provides explicit connectives for reasoning about paths (that can be derived from LTL).

The *serial Horn fragment* of Transaction Logic, i.e., formulas of the form $a_0 \leftarrow a_1 \otimes \ldots \otimes a_n$, plays a special role for *Transaction-Logic programming*, providing a top-down SLD-style proof

procedure. With the above rules, *transactions* can be defined as complex actions, providing a declarative specification of the database evolution. Here, the concept of complex actions is not based on a hierarchical state space but on a sequential one: to execute a_0 (in SLD terminology: to prove a_0) in a state $\mathcal{D}$ means to execute $a_1 \leftarrow a_1 \otimes \ldots \otimes a_n$ by generating n states; thus, the final state $a_0(\mathcal{D})$ is "specified" as $a_1 \otimes \ldots \otimes a_n(\mathcal{D})$. By the connective $\wedge$, dynamic constraints on transaction execution can be specified.

2.6 Labelled Transition Systems and Process Algebras

Process Algebras describe the semantics of processes in an algebraic way, i.e., by a set of elementary processes (carrier set) and a set of constructors.

Formally, their semantics is expressed in terms of *Labelled Transition Systems (LTS)* which are one of the fundamental concepts for modelling processes (cf. [Plo81], [vBB95]):
A *labelled transition system (LTS)* is a triple $(S, A, \rightarrow)$ such that

1. S is a (non-empty) set of states/configurations,
2. A is a (non-empty) set of actions/labels,
3. for every $a \in A$, $\rightarrow^a \subseteq S \times S$ is a binary relation.

An LTS can be equivalently defined as a tuple $(S, \{R_a \mid a \in A\})$ [vBB95], leading to Kripke structures for *polymodal* logics. Note that *finite* LTSs are equivalent to finite automata.

The semantics of a process algebra can either be given as *denotational semantics*, i.e., by specifying the denotation of every element of the algebra (e.g., CSP (Communicating Sequential Processes), [Hoa85]), or as an *operational semantics* by specifying the *behavior* of every element of the algebra (e.g., CCS (Calculus of Communicating Systems), [Mil83], [Mil89]). Here, concepts related to CCS are used.

Basic Process Algebra (BPA). For a given set Act of atomic actions,

$$BPA_{\mathsf{Act}} = \langle \mathsf{Act}, \{\bot, +, \cdot\} \rangle$$

is the basic algebra – i.e., containing the least reasonable set of operators – for constructing processes over Act. $\bot$ is a constant denoting a deadlock, $+$ denotes alternative composition, and $\cdot$ denotes sequential composition: if x and y are processes, then $x+y$ and $x \cdot y$ are processes (syntax and semantics are formally introduced later on with CCS). PA is the union of the concepts of *Basic Parallel Processes* [Chr93] and *context-free processes*.

BPA_{rec} extends BPA with recursion: if x is a BPA_{rec}-process (containing a free variable X), then $recX.(x)$ is a BPA_{rec}-process. The semantics of BPA, BPA_{rec} and extensions (e.g., CCS) is given as an operational semantics in terms of a *labelled transition system*:

Calculus of Communicating Systems (CCS). CCS extends BPA by more expressive operators. The carrier set of a CCS algebra [Mil83, Mil89, Mil90] is given by a set Act of action names from which processes are built by using several connectives. Every element of the algebra is called a *process*. By carrying out an action, a process changes into another process. As an LTS, a process can be regarded as a state or a configuration. Action names become labels and the transition relation is given by the rules specifying the execution of actions.
A CCS algebra with a carrier set Act is defined as follows:

1. With X a variable, X is a process expression.
2. Every $a \in \mathsf{Act}$ is a process expression.

3. With $a \in \mathsf{Act}$ and P a process expression, $a : P$ is a process expression (prefixing; sequential composition).
4. With P and Q process expressions, $P \times Q$ is a process expression (parallel composition).
5. With I a set of indices, $P_i : i \in I$ process expressions, $\sum_{i \in I} P_i$ is a process expression (alternative composition).
6. With $A \subseteq \mathsf{Act}$ an action and P a process expression, $P \restriction A$ is a process expression (restriction to a set of visible actions).
7. With I a set of indices, X_i variables, P_i process expressions, $\mathsf{fix}_j \vec{X} \vec{P}$ is a process expression (definition of a communicating system of processes). The fix operator binds the variables X_i, and fix_j is one of the $|I|$ processes which are defined by this expression.

The fix operator can be omitted if defining equations of the form $Q := P$ are allowed, where Q is a new process identifier and P is a process expression. Process expressions not containing any free variables are *processes*.
The (operational) semantics of a CCS algebra is given by transition rules:

$$a : P \xrightarrow{a} P \quad , \quad \frac{P_i \xrightarrow{a} P}{\sum_{i \in I} P_i \xrightarrow{a} P} \text{ (for } i \in I) \quad , \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{ab} P' \times Q'} ,$$

$$, \quad \frac{P \xrightarrow{a} P'}{P \restriction A \xrightarrow{a} P' \restriction A} \text{ (for } a \in A) \quad , \quad \frac{P_i\{\mathsf{fix}\ \vec{X} \vec{P} / \vec{X}\} \xrightarrow{a} P'}{\mathsf{fix}_i \vec{X} \vec{P} \xrightarrow{a} P'} .$$

Additionally, there are some derived operators and constants:

$$\begin{array}{lcl} \mathbf{0} & := & \sum_{\emptyset} P_i \\ P_1 + P_2 & := & \sum_{i \in \{1,2\}} P_i \\ \partial P & := & \mathsf{fix}\ X(1 : X + P) \quad , \quad X \text{ not free in } P \\ P_1 | P_2 & := & P \times \partial Q + \partial P \times Q \end{array}$$

with the corresponding transition rules

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \quad , \quad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} P'} \quad , \quad \partial P \xrightarrow{1} \partial P \quad , \quad \frac{P \xrightarrow{a} P'}{\partial P \xrightarrow{a} P'}$$

$$\frac{P \xrightarrow{a} P'}{P | Q \xrightarrow{a} P' | Q} \quad , \quad \frac{Q \xrightarrow{a} Q'}{P | Q \xrightarrow{a} P | Q'} \quad , \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P | Q \xrightarrow{ab} P' | Q'}$$

With this definition, the subalgebra $(\mathsf{Act}, \times, 1)$ forms a commutative monoid.

In CCS and related concepts, such as CSP [Hoa85] and ACP [BK85], there is no explicit notion of states, the properties of a state are given by the (sequences of) actions which can be executed.

The idea of CCS specification is illustrated by a storage bit s which can be read and written is described by the process

$$\mathsf{fix}_1 X_0, X_1\ ((\mathsf{output}\ 0) : X_0 | (\mathsf{input}\ 1) : X_1\ ,\ (\mathsf{output}\ 1) : X_1 | (\mathsf{input}\ 0) : X_0) .$$

This means, that there are two processes (components): P_0 which is able to output arbitrary many 0's, and, if written a 1, changes to P_1; and P_1 which can output arbitrary many 1's, and

changes into P_0 if a 0 is written. The storage bit is described by the first one, i.e., P_0. If the process can perform an $\mathsf{output}\ 0$ action, this corresponds to a state (an interpretation) where $\mathcal{I}(s) = 0$, analogous for 1.

Note that the origin of the notion of *bisimulation* (as used in the Xcerpt language for querying XML [BS02]) comes from process algebras.

2.7 Event Languages

Reactive rules cannot only be specified by reactions on atomic, primitive events, but can also use the notion of *complex events*, e.g., "when E_1 happened and the E_2 and E_3, but not E_4 after at least 10 minutes, then do A". Complex events are usually expressed in terms of an *event algebra* by connectives and operators; they can be parameterized by elements of the universe. In contrast to process algebras, where most of the proposed concepts share the same set of operators and constructors, for complex events, there are quite different approaches. Usually, when defining an event language or an event algebra, an algorithm for *detecting* such events is also given for triggering active rules in ECA-like approaches.

In Action Logic (ACT) [Pra90], events are characterized by extended regular expressions. In addition to the usual operators, two implications, preimplication and postimplication are introduced.

2.7.1 The Event Algebra of [CKAK94]

In [CKAK94], an event algebra which is used for event detection in the context of ECA-rules ("on ⟨event⟩ if ⟨condition⟩ do ⟨action⟩") in active databases is proposed. Semantically, an event is a predicate $E : T \to \{\mathsf{true}, \mathsf{false}\}$ where T denotes a set of time instances. For a given set of elementary events, the set of events is defined inductively:

- If E and F are events, then $E \nabla F$ and $E \triangle F$ are events.
- If $E_1, \ldots, E_n$ are events and $m < n \in \mathbb{N}$, then $\mathsf{ANY}(m, E_1, \ldots, E_n)$ is an event.
- If E and F are events, then $E; F$ is an event.
- If E_1, E_2 and E_3 are events, then $A(E_1, E_2, E_3)$ and $A^*(E_1, E_2, E_3)$ are events.
- If E_1, E_2 and E_3 are events, then $\neg(E_1)[E_2, E_3]$ is an event.

The semantics of composite events is defined as follows:

(1) $(E \nabla F)(t) :\Leftrightarrow E_1(t) \vee E_2(t)$,

(2) $(E \triangle F)(t) :\Leftrightarrow E_1(t) \wedge E_2(t)$,

(3) $(E_1; E_2)(t) :\Leftrightarrow \exists t_1 \leq t : E_1(t_1) \wedge E_2(t)$,

(4) $\mathsf{ANY}(m, E_1, \ldots, E_n)(t) :\Leftrightarrow \exists t_1 \leq \ldots \leq t_{m-1} \leq t,\ 1 \leq i_1, \ldots, i_m \leq n$ pairwise distinct s.t. $E_{i_j}(t_j)$ for $1 \leq j < m$ and $E_{i_m}(t)$,

(5) $\neg(E_2)[E_1, E_3](t) :\Leftrightarrow E_3(t) \wedge (\exists t_1 : E_1(t_1) \wedge \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg(E_2(t_2) \vee E_3(t_2))))$,

(6) $A(E_1, E_2, E_3)(t) :\Leftrightarrow E_2(t) \wedge (\exists t_1 : E_1(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg E_3(t_2)))$,

(7) $A^*(E_1, E_2, E_3)(t) :\Leftrightarrow \exists t_1 \leq t : E_1(t_1) \wedge E_3(t)$, when this event occurs, a specified action for every occurrence of E_2 has to be executed in t.

The constructs ∇ (or), $\triangle$ (and) are standard and straightforward. "$(E_1; E_2)$" denotes the successive occurrence of E_1 and E_2, where in case that E_2 is a complex event, it is possible that subevents of E_2 occur *before* E_1 occurs. ANY denotes the occurrence of m events out of n in arbitrary order, which is also expressible by a special ∇-$\triangle$-;-schema. (5) is a complex event which detects the non-occurrence of E_2 in the interval between E_1 and the next E_3. (6) is an "aperiodic" event which is signaled whenever E_2 occurs after E_1 without E_3 occurring in-between. Note that $\neg(E_2)[E_1, E_3]$ occurs with the *terminating* event E_3 whereas $A(E_1, E_2, E_3)$ occurs with *every* E_2 in the interval (if E_3 never occurs, the interval is endless). The "cumulative aperiodic event" (7) occurs with E_3 and then requires the execution of a given set of actions corresponding to the occurrences of E_2 in the meantime. Thus, it is not a simple event, but more an active rule, stating a temporal implication of the form "if E_1 occurs, then for each occurrence of an instance of E_2, an event must occur later which corresponds to the execution of a specified action".

2.7.2 The Characterization of [Sin95]

In [Sin95], another algebra for intertask dependencies is proposed from a workflow modelling point of view, based on [Pra90]. Here, the aim is to schedule a set of pending events such that all of them occur. For a given set Σ of (propositional) elementary events,

- with $\Gamma := \{e, \bar{e} \mid e \in \Sigma\}$, every element of Γ is an event expression,
- $\bot$ and ∂ are event expressions, and
- with E_1 and E_2 event expressions, $E_1 \cdot E_2$, $E_1 + E_2$, and $E_1 | E_2$ are event expressions.

The semantics of event expressions is defined in terms of *traces*. The set $\mathcal{U} := \Gamma^* \cup \Gamma^\omega$ of traces is defined as the set of all finite and infinite sequences over Γ. Then, the *denotation* of event expressions is defined as follows:

$$\begin{array}{ll} [\![e]\!] := \{\tau \in \mathcal{U} \mid \tau \text{ mentions } e\} \text{ for } e \in \Gamma\,, & [\![E_1 \cdot E_2]\!] := \{\upsilon\tau \mid \upsilon \in [\![E_1]\!] \text{ and } \tau \in [\![E_2]\!]\}\,, \\ [\![\bot]\!] := \emptyset\,, & [\![E_1 + E_2]\!] := [\![E_1]\!] \cup [\![E_2]\!]\,, \\ [\![\partial]\!] := \mathcal{U}\,, & [\![E_1 \mid E_2]\!] := [\![E_1]\!] \cap [\![E_2]\!]\,. \end{array}$$

Note that this definition does not yet guarantee that $[\![e + \bar{e}]\!] \neq [\![\partial]\!]$ and $[\![e \mid \bar{e}]\!] \neq \emptyset$. Instead, this is guaranteed by additional restrictions on *legal* computations: In a legal trace,

- every event $e \in \Gamma$ excludes its complementary event $\bar{e}$,
- an elementary event (or its complementary event) occurs at most once in any computation, and
- an elementary event or its complement occurs eventually in each computation.

Note that with this, an "event" $\bar{e}$, occurring *somewhere* in a computation, implicitly restricts the occurrence of e along the *whole* computation.

An interesting property of this approach is the built-in event-detection algorithm via *residuation*: If an elementary event e occurs, all pending events are residuated wrt. e and the remaining pending event is computed:

$$\begin{array}{ll} \bot/e := \bot\,, & (e \cdot E)/e := E\,, \\ \partial/e := \partial\,, & (e' \cdot E)/e := \bot \text{ if } e \text{ occurs in } E\,. \\ E/e := E \text{ if } e, \bar{e} \text{ do not occur in } E\,, & (E_1 + E_2)/e := (E_1/e + E_2/e)\,, \\ E/e := \bot \text{ if } \bar{e} \text{ occurs in } E\,, & (E_1 \mid E_2)/e := (E_1/e \mid E_2/e)\,. \end{array}$$

In [Sin96], the issue of scheduling a workflow satisfying a set of given dependencies based on residuation is addressed. There, the formalism is extended with temporal operators $\Box$ and $\Diamond$:

- with E an event expression, $\Diamond E$, and $\Box E$ are event expressions.

Let $u = (u_1, u_2, \ldots)$ be a trace over Γ and $i \in \mathbb{N}$. Then,

$$
\begin{array}{ll}
u \models_i \partial & \text{for every } u, i \ , \\
u \models_i e & :\Leftrightarrow \ \text{there is a } j \leq i \text{ s.t. } u_j = e \ \ (e \in \Gamma) \ , \\
u \models_i E_1 + E_2 & :\Leftrightarrow \ u \models_i E_1 \text{ or } u \models_i E_2 \ , \\
u \models_i E_1 \mid E_2 & :\Leftrightarrow \ u \models_i E_1 \text{ and } u \models_i E_2 \ , \\
u \models_i E_1 \cdot E_2 & :\Leftrightarrow \ \text{there is a } 0 \leq j \leq i \text{ s.t. } u \models_j E_1 \text{ and } (u_j, u_{j+1}, \ldots) \models_{i-j} E_2 \ , \\
u \models_i \Box E & :\Leftrightarrow \ \text{for all } j \geq i : u \models_j E \ , \\
u \models_i \Diamond E & :\Leftrightarrow \ \text{there is a } j \geq i \text{ s.t. } u \models_j E \ .
\end{array}
$$

Here, if an event is satisfied in some state, it is satisfied in every later state as well, thus, for all *elementary* events $e \equiv \Box e$ (called *stability of events* in [Sin96] – note that $\Diamond e \not\equiv \Box\Diamond e$). $u \models_0 \Diamond E$ is satisfied if there is some i s.t. $u \models_i E$.

2.7.3 The Event Characterization of [LBS99]

The *policy description language* $\mathcal{PDL}$ [LBS99] is an ECA-style framework (see Section 4) for defining "policies". It combines an event description formalism with an action language (see subsequent section). The semantics is defined based on atomic "time steps", called "epochs", in which one or more actual events can be detected (simultaneously).
The combinators are very similar to the above ones. *Basic* events are defined from event literals $e_1, \ldots, e_n$. A basic event occurs in a single epoch:

- if e_i are events, then $e_1 \& \ldots \& e_n$ is an event (conjunction, simultaneous occurrence of the e_is),
- if e_i are events, then $e_1 | \ldots | e_n$ is an event (disjunction, occurrence of at least one of the e_is).

The the event $!e$ is "detected" in an epoch, if e is not detected in this epoch. Using deMorgan rules, $!e$ is defined for all basic events.
Composite events that are detected over several epochs (i.e., in the last epoch where one of its components is detected) are then constructed from composite events $E_1, \ldots, E_n$:

- sequence: $E_1, \ldots, E_{n-1}, E_n$ occurs, if E_n occurs, and has been immediately preceded by E_{n-1} etc.,
- iteration: $\hat{}E$ represents the sequence of zero or more occurrences of E,
- The $group(E)$ construct is used for "ignoring" repeated detections of an event, i.e., the event $(e_1, group(e_2))$ is detected only once in a sequence where $e_1 \ldots e_2 \ldots e_2$ occurs [note that in the semantics definition $group(e)$ is only defined for basic events,
- policy-defined events $pde(parms)$ are defined by *triggers*, i.e., if a (complex) event is detected in some situation, and a given condition is satisfied, the policy-defined event $pde(parms)$ "occurs" in the next step, where $parms$ are determined by the condition and parameters of the complex event.

Note that the above definition does not allow specification of conjunction or disjunction of complex events; for them only sequential connectives are defined.

The semantics of events and event detection is then defined based on *event histories*, *traces* of an event in an event history, and minimal histories of an event E.

Evaluation is done then by maintaining all possible distinct partial traces for relevant events E (using an NDFA for event E). In case that a final state of an NDFA is reached, the event is detected and an ECA rule is activated.

2.7.4 Event Detection by Modal Temporal Logic

As an alternative to event languages, events can be expressed as temporal *constraints* using modal temporal logic as described in Section 2.2. In [Cho95c], event detection is done in a similar way to above by bookkeeping about *potentially satisfied* temporal constraints that are expressed as past tense temporal formula. For each constraint (formula), it is incrementally checked if the current state sequence is a prefix of a model of the formula. If finally the formula is satisfied, the event expressed by the formula is detected.

2.8 Action Languages and Situation Calculus

Action languages are formal models that are used for representing action and reasoning about the effects of actions [Rei93, BGP97, GL93, GL98a, GLL$^+$97, GLL$^+$04, GL98b, EFL$^+$04, GL98a, EFL$^+$04] that have been mainly developed in the Knowledge Representation and Reasoning community.

Central to this method of formalizing actions is the concept of a transition system. Here a transition system is simply a labelled graph where the nodes are states (each described by the set of fluents that are true in it) and the arcs are labelled with actions or sets of action.

Usually, the states are first-order structures, where the predicates are divided into static and dynamic ones, the latter called *fluents* (cf. [San94]). Action programs are sets of sentences that define one such graph by specifying which dynamic predicates change in the environment after the execution of an action. Note that Evolving Algebras/Abstract State Machines (see Section 2.4) are actually a special kind of action programs. Usual problems here are to predict the consequences of the execution of a sequence of (sets of) actions, or to determine a set of action implying a desired conclusion in the future (planning).

Moreover, several action query language exist, that allow for querying one such transition system, that goes beyond the simple queries of knowing what is true after a given sequence of actions has been executed (allowing e.g. to query about which sets of actions lead to a state where some goal is true, i.e. planning as in [EFL$^+$04]).

2.8.1 Situation Calculus

The first, and most prominent concept here is the situation calculus (originally in [MH69], reprinted in [McC90], see also [Rei93]).

States (or situations) are elements of the domain, occurring as an argument for distinguished predicates $\mathsf{holds}(p(x), s)$ and $\mathsf{occurs}(a(x), s)$ where p is a predicate of the application domain and a is an action. Events (mainly actions) in a situation produce new situations: $\mathsf{do}\ a(s)$ denotes the situation which is obtained by executing an action a in a situation s. A situation is a first order functional term $\mathsf{do}\ a_n(\mathsf{do}\ a_{n-1}(\ldots(\mathsf{do}\ a_1(s_0))))$, where a_i are actions and s_0 is a constant denoting the initial situation; the values of fluents in s_0 are specified by formulas of the form $\mathsf{holds}(p(x), s_0)$.

Actions are characterized by *preconditions*, e.g.

$$\mathsf{occurs}(a(x), S) \rightarrow \mathsf{holds}(p(x), S)$$

and their *normal effect*, e.g.

$$\mathsf{holds}(p(x), S) \wedge \mathsf{occurs}(a(x), S) \rightarrow \mathsf{holds}(q(y), do\ a(x)(S))$$

describing how an action changes some fluents. The frame problem is solved by adding axioms for assuming that fluents which are not explicitly changed, remain unchanged [Rei93].

There exist different versions of the situation calculus, e.g., the one used in GOLOG [LRL$^+$97], a logic programming language. There, the predicate holds is omitted and the preconditions are characterized by a distinguished predicate, i.e. $\mathsf{Poss}(a(x), s) \equiv \mathsf{holds}(p(x), S)$. In GOLOG, frame axioms are stated explicitly.

Every set of situation calculus formulas has a natural mapping to Kripke structures.

The situation calculus does not provide constructs for verifying temporal properties, neither modalities, nor temporal connectives. Thus the specification of temporal constraints or requirements is not possible.

2.8.2 Action Languages

Language $\mathcal{B}$

The $\mathcal{B}$ language [GLL$^+$97, GL98a] is a generalization of the so-called language $\mathcal{A}$ [GL93](which itself represents the propositional fragment of the ADL formalism [Ped89]). It allows conditional and non-deterministic actions and, unlike $\mathcal{A}$, also for the representation of actions with indirect effects. A program in $\mathcal{B}$ is a set of static and dynamic laws, of the forms, respectively:

$$L \textbf{ if } F$$
$$A \textbf{ causes } L \textbf{ if } F$$

where L is a fluent literal, F a conjunction of literals, and A an action name. A program in $\mathcal{A}$ is as in $\mathcal{B}$ but without static laws. Intuitively a static law states that every possible state satisfying the conjunction F must also satisfy L, and a dynamic law that if F is satisfied when action A occurs then L is true in the immediate subsequent state. Given a sets of static and dynamic laws, a transition system is defined according to these intuition. Basically, states are all interpretations closed under the static laws, and there is an arc from a state s to a state s' with label a iff all Ls of dynamic rules of the form a **causes** L **if** F, where F holds in s, belong to s', and nothing else differs from s to s'. Besides the above briefly described language $\mathcal{B}$, several other extension of the language $\mathcal{A}$ exist. Language $\mathcal{AR}$ [GKL97], as for $\mathcal{B}$, also allows for modelling indirect effects of actions but in this case, instead of static laws, constraints of the form **always** F, where F is a propositional formula, are used. Language $\mathcal{A}_K$ [SB01] further extends $\mathcal{AR}$ for formalizing sensing actions (i.e. actions for determining the truth value of fluents). Another extension of $\mathcal{A}$, to be found in [LBS99], is the language $\mathcal{PDL}$ which is particularly tailored for specifying policies. A survey and comparisons on extensions of $\mathcal{A}$ can be found in [EFPP04].

Language $\mathcal{C}$

As in language $\mathcal{B}$, also in $\mathcal{C}$ [GL98b, GLL$^+$04] statements of the language are divided into static and dynamic laws. The main distinction between $\mathcal{C}$ and $\mathcal{B}$, besides the fact that $\mathcal{C}$ allows for arbitrary formulas to be caused by action (rather than simply literals as in $\mathcal{B}$ and arbitrary formulas as conditions (rather than conjunction only), is that $\mathcal{C}$ distinguishes between asserting that a fluents "holds" and making the stronger assertion that "it is caused", or "has a causal explanation".

A program in $\mathcal{C}$ is a set of static law and dynamic, of the forms, respectively:

$$\textbf{caused } F \textbf{ if } G$$
$$\textbf{caused } F \textbf{ if } G \textbf{ after } U$$

where F and G are propositional formulas with fluent literals, and U is a formula with both fluent literals and action names.

Intuitively, a static law states that the formula G causes the truth of the formula H, and a dynamic rules states that after U, the static rule **caused** F **if** G is in place. The definition of a transition system for $\mathcal{C}$ is based on *causal theories* [GLL$^+$04]. The idea behind causal theories is that something is true iff it is caused by something else. Let us consider a causal theory T and a set of fluents M. Let us further consider the set T_M of formulae defined as follows:

$$T_M = \{F | \textbf{ caused } F \textbf{ if } \ G \ \wedge \ M \models \ G\}$$

We say M is a *causal model* of T iff M is the unique model of T_M. Given any action program P, a state s and a set of actions K, we consider the causal theory T given by the static laws of P and the static laws corresponding to the dynamic laws whose preconditions are true in $s \cup K$. Then there is an arc between s and s' with label K iff s' is a casual model of T. It is worth nothing that in $\mathcal{C}$, contrary to $\mathcal{B}$, fluent inertia is not assumed by default.

Various extensions to $\mathcal{C}$ have recently appeared in the literature. Most prominently, the language $\mathcal{C}$++ [GLL$^+$04] and the language $\mathcal{K}$ [EFL$^+$04]. $\mathcal{C}$++ further allows for multi-valued, additive fluents which can be used to encode resources and allows for a more compact representation of several practical problems. The language $\mathcal{K}$ which allows for representing and reasoning about incomplete states, and for solving planning problems.

For more details on these languages, as well on the implementation of fragments of them in logic programming, see [EFPP04, GL98a, GLL$^+$04].

Chapter 3

Update Languages

In this chapter we give a survey on existing update languages and concepts that are relevant for our project. For the Conventional Web, i.e. today's Web, the content and structure of data sources is of importance and the semantics of data is missing, as detailed in Chapter 1. Thus, updates act on the (extensional) data level, and on the schema level. The vision of the Semantic Web is that of a Web where the semantics of data is available for processing by automated means, including reasoning mechanisms and intensional data. Here, update concepts both for the data level and for the metadata level, and for intensional data and behavior are needed.

In the first section we present several existing update languages for the conventional Web – these will also most probably play an important role as the base layer. Currently, these are mostly pure update languages for (local) semistructured data, whereas updates "on the Web" are not yet supported.

Next, we give an overview of update concepts for the emerging Semantic Web. There, updates on data have then to be specified on the semantic level, i.e. wrt. RDF, and the issues of maintaining and updating ontologies are discussed. We then give an overview of several update languages defined for knowledge bases and logic programming, that allow for updating not only facts (extensional knowledge), but also derivation rules (intensional knowledge) and reaction rules (behavior).

3.1 Update Languages for the Web

In this section we discuss existing approaches to update languages for XML and for the Web. As the research community focussed its work on the development of *query* languages for XML and for the Web, the development of update languages for XML and for the Web has not received much attention so far. Most likely this is the reason why no declarative XML update language has yet become a World Wide Web Consortium's recommendation. Updates on XML data are still in most cases performed on the DOM level.

Most existing update languages support only changes to local XML data sources, as known from SQL. Literally, "updates on the *Web*" would also mean to update remote information (that in course requires to deal with authentication issues).

There are the following proposals for languages that provide update capabilities for XML data: Lorel, XML:DB's XUpdate, two update extensions to XQuery proposed in [TIHW01] and

in [Leh01], XPathLog, XML-RL Update Language, and XChange. Commercial XML-enabled relational databases also provide update functionality for XML contents that is based on SQL.

Usually, update languages are designed as an extension of a query language with update capabilities. At least, an addressing mechanism for selecting parts of XML documents that are to be modified is needed. Thus, most existing proposals for update languages for XML have a common feature: a XPath expression is used to select nodes within the input XML document; the selected nodes are then considered as target of the update operations. (Note that also tree-walking transformational approaches could be used for declaratively modifying XML data – thus, when talking about updates, e.g. as events in active rules, this possibility must be considered.)

Lorel

The Lorel language [AQM+97] was originally designed based on OQL as a query and update language for Stanford's Lore [GMPQ+97] semistructured database system that, in turn, was based on the graph-based Object Exchange Model (OEM). The language supports just simple updates of nodes in the Lorel data graph (i.e. create and delete object names, create a new object, and insert, delete or modify the value of attributes of an object, and to insert, delete or modify relationships/references). There is no explicit deletion operation for objects. Instead a garbage collection approach is taken. The Lorel query language has been migrated to XML data, but the update features were not ported in the process. Transactional properties of complex updates (and transactions at all) are not supported.

XML:DB's XUpdate

XUpdate [XML00] is an update language developed by the XML:DB group[1], its latest language specification was released in late 2000 as a working draft. Note that, at that time, the query languages XPath, XQL, and XML-QL and the transformation language XSLT were already defined, but XQuery did not yet exist. Thus, also the name "XUpdate" is not related to XQuery.

Similar to XSLT, XUpdate is written in XML syntax and makes use of XPath [xpa99] expressions for selecting nodes to be processed afterwards. Simple atomic update operations to XML documents are possible with XUpdate. The XSLT-style syntax of the language makes the programming, and the understanding of complex update programs, very hard. Transactional properties of complex updates (and transactions at all) are not supported.

Several XML database systems implement this language (see an overview at the end of this section).

XQuery Update extensions

A proposal to extend XQuery [xqu01] with update capabilities is presented in [TIHW01]. XQuery is extended with a `FOR ... LET ... WHERE ... UPDATE ...` structure. The `UPDATE` part contains specifications of update operations (i.e. delete, insert, rename, replace) that are to be executed in sequence. For ordered XML documents, two insertion operations are considered: insertion before a child element, and insertion after a child element. Using a nested `FOR...WHERE` clause in the `UPDATE` part, one might specify an iterative execution of updates for

[1] Formerly at `http://www.xmldb.org`, now `http://xmldb-org.sourceforge.net/`

nodes selected using an XPath expression. Moreover, by nesting update operations, updates can be expressed at multiple levels within a XML structure. A transaction-like concept for complex updates (i.e. treating them as a unit, and executing in an all-or-nothing manner) is not explicitly investigated. Alternative techniques for implementing the update operations are presented for the case when XML data is stored in a relational database (i.e. XML update statements are translated into SQL statements [KK00]). This is the only work on update languages that reports on implementation performance. Using three sets of test data (viz. synthesized data with fixed structure, synthesized data with random structure, and real-life data from the DBLP[2] bibliography database), experiments have been carried out in order to compare the techniques proposed for the core update operations of insert and delete.

Update operations very similar to those described in [TIHW01] have been specified and implemented in [Leh01], extended e.g. by means to specify conditional updates. The proposal has been prototypically implemented in Quip[3] which works directly with the XQuery update statements without any use of SQL statements. The solution has been incorporated into Software AG's Tamino[4] product.

XPathLog

XPathLog [May01a, MB01] is a rule-based logic-programming style language for querying, manipulating and integrating XML data. XPathLog can be seen as the migration from F-Logic [KL89], as a logic-programming style language, for semistructured data to XML. It uses XPath [xpa99] as the underlying selection mechanism and extends it with the Datalog-style variable concept. XPathLog uses rules to specify the manipulation and integration of data from XML resources. As usual for logic-programming style languages, the query and construction parts are strictly separated: XPath expressions in the rule body, extended with variables bindings, serve for selecting nodes of XML documents; the rule head specifies the desired update operations intensionally by another XPath expression with variables, using the bindings gathered in the rule body. As a logic-programming style language, XPathLog updates are insertions. There is no explicit replacement or deletion operation in the basic language (but can be extended with appropriate semantics in the same way as for bottom-up Datalog). Transactional properties of complex updates (and transactions at all) are not supported.

In addition to the plain query and update functionality, XPathLog supports class membership, subclasses, schema data, and nonmonotonic inheritance as known from F-Logic. XPathLog is implemented in the LoPiX[5] system, which, in turn, is based on the Florid[6] system.

XML RL Update Language

The XML-RL Language [LLW03] incorporates some features of object-oriented databases and logic programming. The XML-RL Update Language extends XML-RL with update capabilities. The query and construction parts are strictly separated. The rule body specifies queries to XML documents and the rule head specifies the XML data to be constructed.

Five kinds of update operations are supported by the XML-RL Update Language, viz. `insert before`, `insert after`, `insert into`, `delete`, and `replace with`. Using the built-in

[2]DBLP bibliography, `http://www.informatik.uni-trier.de/~ley/db/`
[3]`http://developer.softwareag.com/tamino/quip`
[4]`http://www.tamino.com`
[5]LoPiX, `http://www.dbis.informatik.uni-goettingen.de/lopix/`
[6]Florid, `http://www.informatik.uni-freiburg.de/~dbis/florid/`

position function, new elements can be inserted at the specified position in the XML document (e.g. insert first, insert second). Also, complex updates at multiple levels in the document structure can be easily expressed. Transactional properties of complex updates (and transactions at all) have not been investigated. No implementation is available.

XChange

XChange [BPS04b, BPS04a] is a declarative language for specifying evolution of data on the (Semantic) Web. The language is currently being developed at the University of Munich and by the Working Group *I5 "Evolution and Reactivity"* of the REWERSE project. XChange builds upon Xcerpt [BS02, BS04], a declarative query and transformation language for the (Semantic) Web. Xcerpt uses a positional or pattern-based selection of data items. A query pattern is like a form that gives an example of the data that is to be selected, like query atoms in logic programming. As in logic programming, a query pattern can contain variables, which serve for retrieving data items from the queried data. The XChange update language uses rules to specify *intensional updates*, i.e. a description of updates in terms of queries. A metaphor for XChange is to see update specifications as forms, where rule head with the variable bindings yields the data terms after update execution.
XChange supports the following features for updating XML data on the Web:

Synchronisation of Updates. XChange provides the capability to specify relations between complex updates and execute the updates synchronously (e.g. when booking a trip on the Web one might wish to book an early flight *and* the corresponding hotel reservation, *or* a late flight *and* a shorter hotel reservation). As the updates are to be executed on the Web, network communication problems could cause failures of update execution. To deal with such problems, an explicit specification of synchronisation of updates is possible with XChange, a kind of control which logic-programming languages lack.

Transactions as Updating Units. Since it is sometimes necessary to execute complex updates in an all-or-nothing manner (e.g. when booking a trip on the Web, a hotel reservation without a flight reservation is useless), the concept of transactions is supported by the language XChange. More precisely, XChange transactions are composed of events (discussed in Section 4.2.1) posted on the Web and updates to be performed on one or more Web sites.
Complex applications specifying evolution of data and metadata on the (Semantic) Web require a number of features that cannot always be specified by simple programs. In XChange transactions can also be used as means for structuring complex XChange programs.

An XChange prototype will be available in the near future, based on the Xcerpt prototype[7].

Commercial and Open Source Systems

Several commercial and open-source database systems provide update capabilities on stored XML data. A comprehensive overview of such systems can be found at `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`. The commercial systems support transactions while the open-source ones in general don't.

"Native" XML Database Systems. The *Apache Xindice* project[8] is a continuation of a project called "dbxml core", implementing XQuery and the XML:DB approach. The open-

[7] Xcerpt Prototype: `http://demo.xcerpt.org`
[8] `http://xml.apache.org/xindice/`

source XML database system eXist[9], and the commercial system X-Hive[10], both support XQuery and XML:DB's XUpdate. The Tamino[11] system (Software AG) does *not* support the XML:DB XUpdate approach, but implements the update functionality from [Leh01] on top of XQuery (that has been prototypically implemented before in Quip[12]). The *eXcelon Information Server* (before: Object Store, now: Progres Software) provides an update functionality that is similar to the XML:DB update proposal, called XUL (*XML Update Language*, extending XSLT); calling such small programs "*updategrams*".

XML-Enabled Conventional Database Systems. At present, most "conventional" (object)-relational commercial database systems also support XML data. Here, XML is stored either directly as VARCHAR, as CLOB, by "shredding", or by "opaque" storage via built-in object-relational datatypes, or in separate files. In either case, XML-specific query and update functionality is provided. Here, XPath as an addressing mechanism is not embedded into XQuery, but into the surrounding SQL via the SQL/XML standard[13] [EM01]. The query part is specified by this standard: XML objects are treated like row objects or column objects in SQL3, and then XML-specific member-methods are applied for evaluating XPath expressions on such data. For the update part, different approaches are currently followed: The MS SQL server provides update functionality via "updategrams", which are similar to eXcelon's XSLT-style XML instances that contain the commands. IBM DB2 and Oracle 10 extend SQL/XML smoothly with update functionality into the SQL/XML syntax [14] by extending to XML the usual SQL `UPDATE ... SET attr=expr`: an update on an XML item x is a function that specifies the updated node inside x (by a relative XPath expression) and the new value. The result of the function is returned and replaces x. Thus, here updates are actually seen as transformations.

Transaction-like concepts

Transaction-like concepts, including the issue of synchronising updates (i.e. specifying complex updates and relations between them, and executing the updates accordingly) are in general only addressed in the larger open-source and commercial systems. For the XChange language, transactional concepts are planned as described above.

Updates on the Web

In contrast to relational languages like SQL, where queries are stated against a local repository, XML and Web query languages seamlessly query as well local XML data, as remote XML data by using URLs in queries. Nevertheless, in the style of traditional database concepts, most of the above (update) languages work on first sight on local XML data, or databases.

Since all these update languages use answers obtained from queries as a basis for updates, it is clear that in principle update statements can be formulated specifying an update of *remote* data. However, in these languages' realizations this is usually not supported, and in fact the remote data is not updatable and even not accessible for the system that executes the statement.

[9] `http://exist.sourceforge.net/`
[10] `http://www.x-hive.com`
[11] `http://www.tamino.com`
[12] `http://developer.softwareag.com/tamino/quip`
[13] `http://www.sqlx.org`
[14] IBM DB2's XML functionality is based on the Xperanto/Xtables project [CFI+00].

Thus, update languages for the Web will have to deal with communication (and authorization) issues.

3.2 Update Concepts for the Semantic Web

Updates on the Semantic Web should deal with the data level, and also with the metadata level, i.e. with ontologies. At the data level updates on the Semantic Web should be possibly not only on the plain XML level, but also on a semantic level, e.g. RDF.

3.2.1 Updates on the Data Level

For executing updates on the data level of the Semantic Web, first an appropriate query language is required. Since all XML query/update languages be applied to RDF serialized as XML, they can also somehow be regarded as RDF query/update languages. Xcerpt/XChange is applied in this way to RDF/RDFS data in [BFPS04b]. With XPathLog, RDFS can be handled more natively, since it supports class membership, subclasses, signatures, and also nonmonotonic inheritance. However, this cannot yet be seen as "RDF querying" since the central concept of RDF, namely that identifying objects by their URIs, is not used.

An overview of existing RDF query languages can be found at `http://www.w3.org/2001/11/13-RDF-Query-Rules/`. Most of them combine well-known basic constructs (SFW clauses, FLWR clauses, logic-programming rules) with navigation and filtering on the RDF graph. Such an RDF path language is also defined in [PPW04] as *RDFTL (RDF Trigger Language) path expressions* that serve as the basis for their ECA rules (see Section 4.2.2). With its (simple) update statements, this language can also count as an RDF update language.

For the Semantic Web, languages have to provide additional ontology-level reasoning mechanisms. Concerning the REWERSE project, the development of a Semantic Web Query Language will be done in the Working Group I4 "Reasoning-Aware Querying" by extending the Xcerpt language accordingly. Based on this language, the XChange concepts will also be lifted to the Semantic Web level in course of the REWERSE project.

3.2.2 Updates on the Meta Level: Ontology Evolution

On the metadata-level, special-purpose tools for ontology evolution are important for managing evolution of the Semantic Web. Some of the existing tools for ontology evolution and works on updates in Description Logics [BCM+03] are described below. This description, though, is not an exhaustive survey on existing tools for ontology evolution.

Anonymous work proposed in [Sin03]. This work discusses change operations on knowledge bases having ALC [SSS91] as a base description logic. In ALC only concepts and relations (roles) between concepts are specified. The objects (i.e. instances of specified concepts) and their relations are neglected.

In order to have an explicit semantics for each operation that changes the knowledge base, the work focusses on formalizing change operations on ontologies. Thus, a system KB is defined, which represents a knowledge base using ALC, and a set of operations (e.g. for deriving new concepts, or for adding new roles) working on the KB. For each such operation, a precondition and a postcondition are formally specified. The author gives a flavor of a technique for formal operationalisation, thus the set of proposed operations can be changed or extended to cope with the actual requirements of the concrete applications.

It is also discussed how the proposed system can be used to model the ontology life-cycle management, dealing with ontology version management. Issues like ontology mapping and ontology merging are neglected but mentioned as future work. Also, multi-user management and transaction management are currently missing.

Anonymous work proposed in [KN03]. A framework is proposed in [KN03] that integrates several sources of information about ontology evolution. The ontology change information can be represented in different formalisms. For a new version of an ontology the changes can be represented as e.g. a log of changes applied to the old version of the ontology, or just as the old and new versions of the ontology. The proposed framework relates the change information that is available in different formalisms, and provides mechanisms to derive new change information from existing information gathered from different sources.

In this work, an ontology of change operations for the OWL [owl04] knowledge model is used as a common language for the interaction of framework tools and components. This ontology of change operations is also used in the OntoView [KFKO02] system (see below). The ontology of change contains basic change operations and an extension that defines complex change operations. A number of rules and heuristics are proposed for obtaining complex change operations from a set of basic operations. The authors want to experiment with these heuristics in order to test their effectiveness and to determine the optimal values for the parameters.

OntoView. OntoView [KFKO02] is a web-based system that currently supports RDF-based ontology languages, like DAML+OIL[15] and provides capabilities like finding, specifying, and storing relationships between ontology versions. It maintains information about the descriptive metadata (e.g. the date of a change), the conceptual relationships (i.e. the logical relationships between constructs in two versions of the ontology), and the transformations between the ontology specifications (i.e. a list of change operations).

OntoView has been inspired by the Concurrent Versioning System, CVS [Ber90], which is used in software development. The implementation of the OntoView system has initially been based on CVS, but the authors want to shift to a new implementation that will be build on a solid storage system for ontologies, like Sesame[16].

Anonymous work proposed in [RSS02]. An interesting proposal can be found in [RSS02], which introduces update semantics into a Description Logics system. The authors argue that this problem is strongly related to the view management problem in databases.

Two kinds of assignments are considered in this work: concept assignment (i.e. expressing properties of the form *a specified object is an instance of a concept*), and attribute value assignments. From our point of view, an important contribution of this proposal is the use of the transaction concept. Two types of transactions are considered: update transactions, i.e. the standard transactions in databases, and completion transactions that revise the existing information.

A transaction may (and generally does) contain two parts: the first part specifies elementary updates, and the second part specifies constraints, i.e. concept assignments. In transaction updates, independence of roles (i.e. concept assignments do not change the values of roles) is assumed and explicit role updates are required.

Implementation issues and optimization proposals are also discussed in [RSS02].

[15] DAML+OIL, `http://www.daml.org/language/`
[16] Sesame, Demo, `http://sesame.administrator.nl`

3.3 Logic Programs Updates

Most of the work conducted until recently in the field of logic programming for knowledge representation has focused on representing static knowledge, i.e. knowledge that does not evolve with time. To overcome this limitation, in recent years, work has been developed in the area of logic programming to deal with updates of logic programs by logic programs, where both fact and logic program's rules can be updated [ABBL04, ABLP02, ALP+00, APPP02, EFST02a, EFST02b, EFST01, Lei03, LP98, MT94, SI99, ZF98].

In our view of the (Semantic) Web as a living organism combining autonomously evolving (rule-based) data sources, declarative languages and mechanisms for specifying its maintenance and evolution are needed. For example, for changing the behavior of a data source, so that a new rule becomes into effect, one should not per force have to be concerned with the complex, interrelated, and dynamically obtained knowledge, and should have a way to simply specify what knowledge is to be changed. This requires the existence of a language for exacting such changes (or updates), which takes in consideration the addition/deletion and change of rules, thereby automating the task of dealing with inconsistencies arising from those updates. The above mentioned works exactly aim at defining the meaning of a knowledge base that is made of a set of rules (or, more precisely, a logic program) after it is subject to a sequence of updates, and where each update is a set of either an external events or the additions, deletions of rules. In this Section we briefly review some of these approaches for updating of logic programs. For a more extensive review see e.g. [EFST02a, Lei03].

3.3.1 Updates of Logic Programs

As the world changes so must programs that represent knowledge about it. When dealing with modifications to a knowledge base represented by a propositional theory, two kinds of abstract frameworks have been distinguished both by Keller and Winslett in [KW85] and by Katsuno and Mendelzon in [KM91]. One, theory revision, deals with incorporating new knowledge about a static world state. The other deals with changing worlds, and is known as theory update. In this section, we are concerned with theory update only, and, in particular, within the framework of logic programming.

In [MT94], the authors introduced a logic programming language for specifying updates to knowledge bases, which they called *revision programs* and here, to avoid confusion with theory revision, we refer to as MT-revision-program. Given the set of all models of a knowledge base, a MT-revision-program specifies exactly how the models are to be changed.[17]

The language of MT-revision-programs is quite similar to that of logic programming: MT-revision-programs are collections of *update rules*, which in turn are built of atoms by means of the special operators: $\leftarrow$, *in*, *out*, and ",". The first is an implication symbol, *in* specifies that some atom is added to the models, via an update, *out* that some atom is deleted, and the comma denotes conjunction. I.e. update rules are of the form:

$$\begin{aligned} in(p) &\leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \quad \text{or} \\ out(p) &\leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \end{aligned}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are atoms, and $m, n \geq 0$. These rules state that if all the q_is, and none of the s_is, are in the knowledge base then p should be in (resp. not be

[17] For more detailed motivation and background the reader is referred to [MT94, PT95].

in) the knowledge base. It is worth noting here some similarities between these update rules and STRIPS operators [FN71], in that both specify what should be added and what should be deleted from the current knowledge base. However, differently from STRIPS the preconditions of these rules may depend on the models of the resulting knowledge base. With STRIPS they may only depend on the models of the previous knowledge base.

In [PT95], a correct transformation of MT-revision-programs into logic programs is defined, that immediately provides a means for implementing the latter. This transformation simply amounts to deleting all the *in*s from the above rules, and to replacing the *out*s by logic programming negation. This requires considering extensions of logic programs that deal with negation in rule heads, such as extended [GL90] or generalized [LW92] logic programs. Moreover additional rules, catering for the inertia of atoms, must be added. As such, MT-revision-programs are simply regarded as logic programs that update models of a knowledge base or, in other words, as logic programs that update sets of facts.

In [LP98] it is noted that to deal with sequences of updates of logic programs, besides considering inertia of atoms, inertia of rules should also be taken into account. To cope with sequences of updates of logic programs, [ALP$^+$00] defines the framework of Dynamic Logic Programming (DLP). In DLP, sequences of generalized programs $P_1 \oplus \ldots \oplus P_n$ are given. Here a generalized logic program is a program where default negation may appear both in rule bodies or heads. Intuitively a sequence is to be viewed as the result of, starting with program P_1, updating it with program P_2, ..., and updating it with program P_n. The role of dynamic programming is to ensure that the newly added rules (from latter programs) are in force, and that previous rules (from previous programs) are still valid (by inertia) as far as possible, i.e. they are kept for as long, and for each objects, as they do not conflict with newly added ones. For example, consider that a logic program with a rule $a(X) \leftarrow b(X)$ is updated with the rule $not\, a(X) \leftarrow c(X)$. It is up to DLP to guarantee that after the update, all objects satisfying b also satisfy a, unless they satisfy c. In this very simple case this would amount to replacing the original rule by $a(X) \leftarrow b(X), not\, c(X)$.

The semantics of dynamic logic programs is defined according to this rationale. Given a model M of the last program P_n, start by removing all the rules from previous programs whose head is the complement of some later rule with true body in M (i.e. by removing all rules which conflict with more recent ones). All others persist through by inertia. Then, to properly deal with default negation, add facts $not\, A$ for all atoms A which have no rule whose body is satisfied in M, and compute the least model. If M is a fixpoint of this construction, M is a stable model of the sequence up to P_n.

A comparative study on sequences of updates of logic programs can be found in [EFST02a]. There, a syntactic redefinition of DLPs is presented, and semantical properties are investigated. In particular, studies on the DLP-verification of well known postulates of belief revision [AGM85], iterated revision [DP97], of theory updates [KM91] have been carried out. Further structural properties of DLPs, when viewed as nonmonotonic consequence operators, are also studied in [EFST02a]. Structural properties of logic program updates are also studied in [DDDS99]. Another important result of [EFST02a] is the clarification of the close relationship between DLPs and inheritance programs [BFL99]. Though defined with different goals, inheritance programs share some close similarities with DLP. Inheritance programs [BFL99] aim at adding inheritance to disjunctive logic programming with strong negation.

Other approaches to updates of logic programs by logic programs are presented in [IS95, IS03, SI99]. Based on an abductive framework for (non-monotonic) auto-epistemic theories, that make use of the notion of negative explanation and anti-explanation, in [IS95] the authors

define "autoepistemic updates". Based on this work, in [SI99] they employ this new abduction framework (in this case rewritten for logic programming instead) to compute minimal programs which result from updating one logic program by another. In their framework, several updates are possible because non-deterministic contradiction removal is used to revise inconsistencies (through abduction) between an initial program and the one updating it, giving preference to the rules of the latter. In their framework, updating and revision take place simultaneously.

Yet another, independently defined, approach to logic programs updates, can be found in [ZF98]. As in DLPs, the semantics of the update of a program by another is obtained by removing rules from the initial program which "somehow" contradict rules from the update program, and retaining all others by inertia. Additionally, at the end, prioritized logic programs are used to give preference to rules from the update program over all retained rules of the initial program. For extended comparisons with the various approaches to sequences of updates of logic programs see [Lei03].

3.3.2 Logic Programming Update Languages

Dynamic logic programming does not by itself provide a proper language for specifying (or programming) changes to logic programs. If knowledge is already represented by logic programs, dynamic programs simply represent the evolution of knowledge. But how is that evolving knowledge specified? What makes knowledge evolve? Since logic programs describe knowledge states, it seems that logic programs could describe transitions of knowledge states as well. It is natural to associate with each state a set of transition rules to obtain the next state. As a result, an interleaving sequence of states and rules of transition will be obtained. Imperative programming specifies transitions and leaves states implicit. Traditional logic programming could not specify state transitions. With the language of dynamic updates LUPS [APPP02], both states and their transitions are made declarative.

LUPS update programs are sequences of update commands that state which logic program's rules are to be added or deleted, and in which conditions. Example of such LUPS commands are:

$$\textbf{always } L_0 \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_n$$
$$\textbf{retract } L_0 \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_n$$

The first command above states that, from the moment it is given onwards, whenever $L_{k+1}, \ldots, L_n$ are true the rule $L_0 \leftarrow L_1, \ldots, L_k$ should be added, whereas the second states that, when $L_{k+1}, \ldots, L_n$ the rule should be deleted. A declarative meaning of a LUPS update program is defined by first obtaining a sequence of logic programs with the additions and deletions of rules, and then obtaining the semantics of such a sequence by using DLP.

Note the similarities of these LUPS update commands with Event-Condition-Action (ECA) rules. However there are two distinctive features. Unlike in ECA rules, in LUPS there is no distinction between events and conditions. In other words, in LUPS only the evaluation of conditions triggers the actions of updating the program. Another distinction is in what regards the actions. In LUPS, actions are limited to updates of the program. But, contrary to what happens in most ECA formalisms, these updates can be besides the insertion or deletion of facts, also the insertion or deletion of rules. This way, LUPS is capable of updating the (derivation) rules of a knowledge base.

To deal with external events, in [EFST01, EFST03] an extension to LUPS, called EPI, is defined. With EPI, commands resemble more the ECA rules, in which an update is triggered by events and evaluation of conditions.

It is clear that LUPS and EPI commands can be seen as reaction (or behavioral) rules. In a dynamic environment, such as the one described in Chapter 1, not only the fact and derivation rules may be subject to updates, but the very reaction rules should be allowed to be changed. Both LUPS and EPI allow for the update of facts and derivation rules, but none of them allows for updates in the reaction rules. KABUL [Lei03] is an extension of LUPS that is capable of dealing with some forms of updates to reaction rules by allowing, in the commands such as the ones above, the replacement of a logic programming rule by a LUPS command. This way, one can specify that, under some conditions, a LUPS command is to be added (or issued) or deleted (or stop to be active).

The EVOLP framework [ABLP02] appeared in the line of development of the above languages for specifying and programming the evolution of knowledge bases represented as logic programs. Distinctly from these extant languages, which introduce a lot of new programming constructs, each encoding a high level behavior of addition and deletion of rules, EVOLP's up front goal was to enable the evolution of logic programs by adding to traditional logic programming as few constructs as possible, i.e. by staying as close as possible to the usual language of logic programs. Basically, EVOLP syntax is simply that of logic programs, with the addition of a special predicate $assert(R)$ whose sole argument is itself a full-blown EVOLP rule (i.e. a logic program rule, or a rule with $assert$s). The meaning of sequences of EVOLP programs is obtained by sets of traces, each trace being a sequence of logic programs whose semantics is defined by DLP. Moreover, EVOLP considers both the evaluation of conditions, and the existence of external events. This way, EVOLP is capable of specifying both derivation rules (in logic programming), and reaction rules that are triggered by events, evaluate conditions and update the derivation rules (as in ECA rules), as well as rules that cater for the update of reaction rules themselves.

A systematic approach for describing, classifying, and reasoning about knowledge base updates is presented in [EFST02b]: an *evolution frame* parameterizes the update mechanism and the semantics of the knowledge base. It consists of a set of (generic) *update actions* (e.g., insert, deletion, and change of rules), an *update policy* how such actions are executed (e.g., directly, or after considering possible inconsistencies), a *realization assignment* that characterized how the actions are actually executed on the rule base, and a semantics for actually evaluating the rules of the knowledge base.

Chapter 4

Activity and Reactivity in Databases

This chapter presents a state-of-the-art survey on active and reactive behavior in databases, especially, Event-Condition-Action (ECA) rules. Briefly, ECA rules have the following semantics: when an event occurs, evaluate a condition, and if the condition is satisfied then execute an action. In the literature, ECA rules are also referred to as triggers, active rules, or reactive rules. This survey starts by tackling the issue of active database systems, and then gives an overview of some recent proposals of ECA rules for semistructured data, for the Semantic Web, and the non-ECA approach of Active XML.
Conceptually, an ECA rule concept consists of several parts:

- Event part: ECA rules can either react on atomic events, or they can use languages for specifying complex events. Such *event languages* have been described in Section 2.7. The event specification language must be accompanied by a suitable *event detection algorithm*.
- Condition part which is usually based on a query language.
- Action part (in general, a kind of a small program) that specifies the actions to be executed.
- ECA execution model: this includes different possibilities for how the ECA rule is applied (before or after or deferred, statement-oriented or set-oriented, its transactional embedding etc), and policies of the ECA engine (e.g. for conflict resolution).

Depending on the choice of the above sublanguages, a broad range of behaviors can be designed. ECA languages based on atomic events are e.g. used for maintaining consistency (as in the well-known SQL triggers) in course of execution of a surrounding process. On the other end of the range, ECA languages that allow for complex events can themselves be used for *specifying* the behavior of a system in a rule-based way (cf. Action languages as described in Section 2.8), up to the definition of rule-based agents (see Section 5).

4.1 ECA rules in Active Database Systems

ECA rules were first investigated by the database community in the late 1980s within the context of active database management systems. Briefly, an active database management system (ADBMS) is a database management system that supports ECA rules. Support for ECA rules

in database systems was initiated in the late 1980s and extensively explored during the 1990s, for example:

- Several working prototypes have been implemented, to name a few, ACOOD [BL92], Ariel [Han96], EXACT [DJ97], NAOS [CCS94], Ode [GJ92], Postgres [PS96], Reach [BZBW95], Samos [GD94], Sentinel [CAMM94], and TriGS [KR98]. More than twenty suggested active database prototypes have been identified.
- Seven workshops have been held between 1993 and 1997, viz. RIDS [PW94, Sel95, GB97], RIDE-ADS [WC94], a Dagstuhl Seminar [BCD94] and ARTDB [BH96, AH98].
- Two special issues of journals [Cha92, CW96] have been published.
- Two text books [WC96b, Pat99] and an ACM Computing Survey [PD99] have been written.
- Most relational database systems support simple forms of ECA rules, e.g. SQL triggers.

In addition, the groups within the ACT-NET consortium[1] reached a consensus on what constitutes an active database management system with the publication of the Active Database System Manifesto [ACT96].

The number of ADBMS publications peaked around 1994-1997 with roughly 40-50 publications each year. During the late 1990s most active database groups moved their research in other directions, e.g., bioinformatics, data warehousing, data mining, and semistructured data. Since 1998 the number of ADBMS publications have significantly decreased and they are now rare in major journals and conferences. Thus, the ADBMS area is not very active today and the survey done by Paton and Diaz [PD99], and the outcome from RIDS'97 [GB97] still reflect the current state-of-the-art concerning ADBMS. Instead, the work on ECA rules is picked up by other communities, e.g., AI [LBS99], and semistructured data (see later sections in this chapter).

An ADBMS can be described by its knowledge model and execution model. The knowledge model describes what can be said about the ECA rules, for example, what type of events (primitive, composite) are supported, in which context the rule condition is evaluated with respect to the database state (e.g., the database state at the beginning of the transaction). The execution model describes how the ECA rules behave at runtime, for example, when the condition is evaluated with respect to the triggering event (i.e., coupling modes), scheduling of rules, triggering of multiple rules.

Briefly, there are two approaches for adding ECA rule support to a DBMS: the layered approach and the integrated approach. The layered approach adds ECA rule support on top of an existing DBMS, whereas the integrated approach embeds ECA rule support into the internal components of the DBMS. Although the layered approach might be easier to implement, it restricts what type of ADBMS features that can be implemented and also how efficient they can be implemented. For example, it might be problematic or even impossible to implement support for a certain coupling mode or a transaction event due to the lack of *hooks* to the internals of the DBMS.

Most ADBMSs are monolithic and assume a centralized environment, consequently, the majority of the prototype implementations do not consider distribution issues. Initial work on how ADBMSs are affected by distribution issues is reported in [Sch96, KL98, BKLW99, YC99]. For example, the event signalling must now be specified in terms of its visibility. In other words, is the event signal visible to the entire distributed environment or only visible at the node that generated the event signal?

[1] A European research network of Excellence on active databases 1993-1996.

It is common knowledge that users of computer programs (which may be based on databases) are reluctant to use something that is slow. Thus, performance issues are in most cases of crucial importance for any computer based program, including active database systems.

One of the first indications of the performance for active databases was reported in [DPG91]: With regard to the active DBMS prototype ADAM it was identified that

> " ... the introduction of rules makes programs on average about *twice* as slow as they are when the rule mechanism is disabled."

As of now, only a few concrete experiments have been performed with regard to the performance of active databases [GBLR98, Ker95], the most extensive one being the BEAST benchmark project [GBLR98]. The BEAST project evaluated the performance of four active object-oriented database prototypes with respect to event detection, rule management, and rule execution. In addition, twelve hypotheses concerning ADBMS performance were investigated and verified. From a practical point of view, the outcome of the BEAST project can help designers of ADBMS to avoid design solutions that have been verified as bottlenecks with respect to ADBMS performance.

As of now there are several techniques (or design solutions) available that can be used to enhance the ADBMS performance, for example, rule indexing, dedicated event detectors, and concurrent rule execution. However, performance issues for ECA rules in the context of XML and the Semantic Web are unexplored.

Given the activities that have taken place since the early 1990s within the ADBMS community and the initiated transfer of research results into commercial products, one might consider whether there is anything left to be done on ECA rules? The answer is yes, but not necessarily within the mainstream database field.

In the forthcoming sections we will elaborate on current trends and open research questions concerning ECA rules. In particular:

- Agent technology and active databases,
- Tools and methodologies for designing with active rules,
- ECA rules and XML

4.1.1 Agent Technology and Active Databases

Since rule engines are frequently used by both agent systems and active databases it is not surprising that attempts have been made to integrate active database technology and agent technology.

A comparison between the characteristics of ADBMSs and agent systems was reported in [BGK+95]. It is shown that although they are developed in different fields of computer science, they have much in common. Hence, researchers in both communities can benefit from the exchange of ideas and techniques developed. In particular,

- There are some key differences in language syntax and semantics between the two research fields. Thus, concepts such as powerful event languages can be adopted by the agent community. Similarly, ADBMSs can benefit from incorporating more complex reasoning and deliberation about which action(s) to execute.
- Agent systems will need to adopt database features such as recovery techniques, rollbacks, and history. This is especially important for real world agent systems that need to be able to

recover from failures etc. Indeed the notion of building more safety into agents is currently an active research area [saf04].

The work reported in [AS97] contrasts ADBMSs with respect to the notions of weak and strong agency. It is demonstrated that ADBMSs already support the notion of weak agency (autonomy, social ability, reactivity, pro-activeness), whereas no aspect of strong agency (knowledge, belief, intention, obligation) is yet supported. Although ADBMSs do not support strong agency, there is speculation on how the notion of strong agency can improve the adaptability and flexibility of an ADBMS. According to the authors of [AS97], this can be achieved by incorporating *general purpose reasoning abilities* into an ADBMS.

The work in [BCL97a, BCL97b] focuses on using ECA rules for supporting the major cooperation strategies (task sharing and result sharing) as formulated in distributed artificial intelligence. Briefly, cooperation strategies are mapped down to ECA rules in three steps:

- The first step is to model high-level speech acts by state diagrams. The state diagrams present an overview of how the various speech acts interact, for example, how should an agent react to a received *Accept* speech act.
- The second step is to extract intermediate representation, i.e., classes, algorithms, and high-level ECA rules that are needed in order to capture the semantics of the speech act protocol.
- The final step is to generate ECA rules for a specific ADBMS. Three features are suggested as important for any ADBMS that would like to support advanced collaboration strategies: i) composite events, ii) composite event restrictions, and iii) dynamic event and rule creation.

4.1.2 Tools and Methodologies

Traditionally, software developers rely upon the use of tools and methodologies for developing an information system. Once the information system has been implemented, users such as database administrators use various types of tools such as debuggers and browsers which can facilitate the process of maintaining an information system.

Although active capability in the form of ECA rules is now available in most commercial databases, its usage in practice is low. One of the major reason for this is the lack of proper tools and methodologies for developing ECA rule based software. Developing tools and methodologies for designing with ECA rules has previously been suggested as one of the crucial issues in order to increase the usage of active database technology in practice [Day95, SKD95, BH96, WC96a, GB98, Dia99]. For example the participants at the ARTDB'95 workshop pointed out that [BH96]:

> ".. most software developers do not use active features in their projects due to complexity. Currently, software designers have no guidelines on what should be implemented as ECA rules and what should not. The participants agreed that there is a need to provide tools for supporting active features in the software development process. It was further suggested that active database tools should be integrated with the software design tools that are already in common use."

As of now, much work has been done on developing tools for static and dynamic verification of ECA rules. It is assumed that these tools are mostly to be used during the design phase of the software development process. Unfortunately, most of these tools are developed and used in isolation, i.e. they are not integrated or used together with a real active database system

[GB98]. Thus, research results from these isolated tools may be difficult to transfer to tools that are used together with a real active database system.

In addition to developing tools for the design phase of the software development process, we envision more work on theoretical foundations and tools for supporting the earlier phases (requirements engineering / analysis) of the software development process. For example, what type of tools can be used when software developers derive active rule capabilities from the application requirements. It is envisioned that more work is needed in the field of developing tools and techniques for simulation of active rules applications. These *simulation* tools can be useful for rapid prototyping or for demonstrating active rule semantics.

Although tools for active databases have been developed and used, Diaz concludes in his chapter on tool support for active databases that [Dia99]:

> "Unfortunately, there is not yet a proper methodology that guides throughout the whole process of building active databases."

The perhaps most complete software methodology for designing with active rules is the IDEA methodology [CF97], which also has an extensive set of supporting tools. From a practical perspective, software developers may be reluctant to use the IDEA methodology since it implies that they will have to learn how to use a completely new software methodology, rather than learning how to use an extension to, for example, their existing software methodology.

Although, most researchers agree upon the importance of business rules as one of the main sources from which active rules can be derived, there is still very little work reported on the relationship between business rules and active rules. In our view, business rule modelling takes place during the phases of requirements engineering and analysis. At some point during the analysis or design phase a decision is made, which separates out a set of business rules that are suitable to be implemented as ECA rules and another set of business rules that are not suitable to implemented as ECA rules. To the best of our knowledge there are no guidelines available which can help a software developer to identify which business rules that are suitable to implement as ECA rules.

In addition to the usage of active capability in the database community, active capability in the form of ECA rules has proven to be useful in areas where the database component is not considered as mandatory. Thus, initial work has been done on unbundling active capabilities from a database system [GKvBF98]. Taking this process a step further, applications that are based on ECA rules but not necessarily are built on top of a database will also require guidelines and tools for deriving the proper ECA rules. Hence, what is needed is a methodology (with associated tools) that provides a software developer with guidelines on how to derive ECA rules from user requirements which can then be mapped into executable ECA rules. In such a scenario the choice of using a database system or not for providing active capabilities (or even ECA rules) is reduced to a design and implementation issue. Thus, it is not an issue when extracting potential active capability during the requirements engineering and analysis phase. In summary, the following is needed:

- **Methodologies and notations for designing with ECA rules**, preferably these methodologies should be based on existing common methodologies for software development and use the UML notation [RJB99]. Hence, existing knowledge on software methodologies can be reused, rather than inventing new software methodologies with support for active rules. For example, how can software engineers derive and support active rule capabilities during the software development process using OMT [RBP+91], Booch [Boo94], or ROP [JRB99].

- **Tools for designing with ECA rules**, preferably these tools should be integrated with existing CASE tools and used with real active database systems (research prototypes or commercial systems).

4.2 Reactivity on the Web

This section describes existing work related to reactivity on the Web. Reactivity is expressed by means of Event-Condition-Action rules (also called active rules, triggers, or reaction rules) inspired from active databases [Pat99, WC96b, DG96].

Although ECA rules have been extensively explored within ADBMS, not all active rules applications need a complete DBMS. The idea to unbundle active rules capabilities from an ADBMS has previously been suggested in [GKvBF98] and [GB98]. It is assumed that unbundling active capabilities from an ADBMS is likely to open up the possibilities for:

- use of active capabilities with arbitrary DBMSs,
- use of active capabilities in broader contexts separate from a DBMS, and
- use of active capabilities in heterogeneous environments.

Unbundled ECA rule engines (e.g., ruleCore [rul] and Amit [IBM]) are a necessary step for transferring ADBMS technology to the Semantic Web, since one cannot assume that all Semantic Web applications have access to a fully fledged ADBMS. In addition, ADBMS technology needs modification in order to adapt to the Semantic Web environment, where information is distributed and the data model is far richer than in the relational context.

The semistructured nature of XML data gives rise to new issues affecting the use of ECA rules. These issues are principally linked to the choice of an appropriate language syntax and an execution model.

In the relational model, the granularity of data manipulation events is straightforward, since insert, delete or update events occur when a relation is inserted into, deleted from or updated, respectively. With XML, this kind of strong typing of events no longer exists. Specifying the granularity of where data has been inserted or deleted within an XML document becomes more complex.

Again in the relational model, the effect of data manipulation actions is straightforward, since an insert, delete or update action can only affect tuples in a single relation. With XML, actions now manipulate entire subdocuments, and the insertion or deletion of subdocuments can trigger a set of different events. Thus, the analysis of which events are triggered by an action can no longer be based on syntax alone.

Compared to rules for relational databases, ECA rules for XML data are more difficult to analyze, due to the richer types of events and actions. However, rules for XML have arguably less analysis complexity than rules for object-oriented data. This stems from the fact that object-oriented databases may permit arbitrary method calls to trigger events, and determining triggering relationships between rules may therefore be very difficult. ECA rules for XML, in contrast, can be based on declarative languages such as XQuery, and so are more amenable to analysis. Here, it is an important decision whether the ECA execution should be restricted to modifications executed in the query/update language, or if the rules should also react on manipulations of the XML data by other means (e.g., on DOM level).

Reactivity in the (Semantic) Web context is still quite an open research issue. An overview of the work done in this field is given in the following, as some of the investigated ideas could

play an essential role in realizing reactivity on the Semantic Web.

4.2.1 Event-Condition-Action Rules for XML/Conventional Web

Below, approaches to active rules and ECA rules for XML data and for the Web are described. ECA on the Web comes in different flavors: extending the "local" SQL3 ECA functionality to XML, publish-subscribe systems that monitor changes and send messages, ECA rules integrated in a "controlled" distributed environment, and globalized variants of ECA rules.

ECA Rules based on XSLT, Lorel, and XQuery

Several approaches have been presented that extend the ECA paradigm from SQL to languages of the XML world, following the development of the latter.

XSLT and Lorel. In [BCP00], reactive capabilities are investigated in the context of XSLT [xsl01] and of Lorel (viz. its migration to XML in [GMW99]). Events are considered insertion and deletion of elements, and insertion, deletion, and update of attribute and text nodes. The authors discuss the problem of detecting changes to an XML document. This problem is more intricate than in SQL databases since XML documents can be manipulated either by their DOM interface, by XSLT transformations (i.e. by replacing the document against a slightly changed one), or by high-level update languages, and by external tools.

In all these cases, an external rule processor engine is fed with rule specifications. The event is, as usual, an update event on an XML document, or an external event; with the occurrence of an event, a value can be bound to a variable. Condition and action are expressed together. In the XSLT case, this condition-action part is expressed as a sequence of XSLT templates, where the "root" template is applied to the value of the variable bound in the event part. The result of applying this XSLT fragment is returned as the outcome of the rule application for further processing. In the Lorel case, the condition-action part is a usual Lorel `SELECT` or `UPDATE` statement. Similar to SQL3, different granularities (set-oriented and node-oriented) are supported; the functions `old` and `new` are supported to denote the values of the node before and after the update execution; and conflict resolution strategies are discussed. [BCP00] does not mention a distinction between "before" and "after" triggers.

The main guidelines of the implementation of active document systems based on XSL and Lorel are discussed; a prototype of the Lorel version is reported.

XQuery. The approach from [BCP00] is continued for XQuery (with the update constructs from [TIHW01], described in Section 3.1) in [BBCC02] with **Active XQuery**, emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases (and using the same syntax for `CREATE TRIGGER`). *Active XQuery* adapts the SQL3 notions of `BEFORE` vs. `AFTER` triggers and, moreover, the `ROW` vs. `STATEMENT` granularity levels to the hierarchical nature of XML data. The core issue here is to extend the notions from "flat" relational tuples to hierarchical XML data. Thus, XQuery updates on complex nodes are seen as bulk update statements, consisting of several simple updates. Bulk update statements are transformed (i.e. expanded) into equivalent collections of simple update operations. As update statements are expanded, triggers are activated by update operations relative to internal portions of fragments of data (in contrast to the similar approach in [BPW02] that is described below). Again, conflict resolution is discussed. The development of a prototype is planned when XQuery updates will become a W3C recommendation.

Generic ECA for XML. A similar approach to ECA rules for standard XML languages in the style of SQL3 triggers is described in [BPW02] and [PPW03] (here, together with an ECA language for RDF, that is described below in Section 4.2.2). ECA rules are expressed as `on ... if ... do`. Events can be of the form `INSERT e` or `DELETE e`, where `e` is an XPath expression that evaluates to a set of nodes; the nodes where the event occurs are bound to a system-defined variable $*delta* where they are available for use in condition and action parts. An extension for a replace operation is sketched. The condition part consists of a boolean combination of XPath expressions. The action part consists of a sequence of actions, where each action represents an insertion or a deletion. For insertion operations, one can specify the position where the new elements are to be inserted using the `BELOW`, `BEFORE`, and `AFTER` constructors.

Regarding the semantics of the language, an immediate scheduling of rules that have been fired is used. Updates are not immediately executed: the inserted or deleted nodes are annotated, the triggered active rules are evaluated and, at the end of the evaluation, the updates are actually executed. In contrast to the approach of *Active XQuery*, complex updates are treated as atomic. The focus of [BPW02] is on the analysis of rule behavior, i.e. techniques for determining triggering and activation relationships between rules. A prototype, based on flat files, is reported in [PPW03]. This same paper presents some ideas too for the case when ECA rules are distributed on the Web. As future work, the coordination of the evaluation of conditions is proposed in case that the events and actions are considered to occur at the same local peer.

Publish/Subscribe and Notification Systems

Publish/Subscribe and Notification systems deal with a restricted version of ECA rules. Such systems do not support execution of updates to the underlying HTML/XML data. Instead changes to documents are monitored and reported. In most cases, they actually use only a single rule pattern whose event can be configured by a subscriber; the condition is empty, and the action consists of sending a (possibly parameterized) notification.

Here two types can be distinguished: the change detection and notification mechanism is located at the same place as the repository, or a separate service is provided that can be told to monitor remote sources and to notify the customer of changes.

A Publish/Subscribe (pub/sub) system (e.g. see [TRP+04]) receives messages from publishers and notifies subscribers if the messages match the subscriptions. In most cases, the ECA functionality is local to the pub/sub system. The research focus in this area is less on ECA and communication issues, but more on efficient evaluation of a large number of subscriptions on a large number of subscribers. The communication follows a pure *push* pattern, i.e., information (published items) are pushed from their originators to the pub/sub system, and derived information (notification about changes) is pushed from the pub/sub service to its subscribers.

An even simpler technology is followed by repositories with notification functionality, where a restricted ECA functionality is directly implemented in the repository. Subscribers can inform the notification service that they want to be informed about changes in its document and receive notifications. As above, the communication follows a pure *push* pattern, where information (changes) is pushed from the repository to its subscribers.

Another communication paradigm is followed by "global" internet query systems, e.g. NiagaraCQ [CdTW00] based on *continuous queries*. In it, users can "register" queries at the service that then continuously states the query against the source, and informs the user about the answer (or when the answer changes). In these systems, communication combines *pull* and

push: the CQ system *pulls* information from sources, and *pushes* derived information to the end user. Research focus here is again on the efficient evaluation (and combination) of a large number of continuous queries.

Considering the goals of the REWERSE project, notifications and pub/sub functionality will be sufficient in many cases, but it is also intended to use full ECA rules throughout the Web.

Generic Remote ECA Rules

Except for the global continuous query services, all the above approaches are "local", in that, as in SQL3, work on a local database, are defined inside the database by the database owner, and consider local events and actions. In [BCP01], an infrastructure for user-defined ECA rules on XML data is proposed that makes use of existing standards for XML and for (XML-based) communication. Here, rules that should be applied to a repository x can be defined by arbitrary users (using a predefined XML ECA rule markup), and can be submitted to x where they are then executed. The definition of events and conditions is up to the user (in terms of changes and a query to an XML instance). Only the actions are still restricted to sending messages.

The approach implements a subscription system that enables users to be notified upon changes on a specified XML document d somewhere on the Web. For this, the approach extends the server where d is located by a rule processing engine. Users that are interested in being notified upon changes in d submit suitable rules to this engine that manages all rules corresponding to documents on this server. Thus, evaluation of events and rules is local to the server, and notifications are "pushed" to the remote users. Note that the actions of the rules do not modify the information, but simply send a message.
Rules are themselves marked up in XML, i.e., by

```
<event> {insert|delete|update} xpath-expr </event>
<condition> query </condition>
<action> SOAP-method-call </action>
```

where the event part specifies the occurrences of simple update operations on elements that are addressed by *xpath-expr*. The condition part is an (XQuery) query that is interpreted as true if it returns a nonempty answer, and that may refer to the nodes on which the events occurred. This is realized through the variables `old` and `new` that represent the nodes on which the events occurred with their past and current values. The action part contains the call of a SOAP [soa00] method, but restricted to implement the call to a message delivery system that transfers information to specified recipients. It is assumed that complex parameters can be passed to the SOAP method that is invoked.

In the basic scenario, such rules are specified by users, and sent to an XML-ECA-enabled server that will process them and will, accordingly, execute the SOAP method that sends a notification message to the subscriber. In this case, the user must know where the events will occur. In a more complex scenario, there is a *rule broker* that even manages the generation of rules, and their allocation with a suitable repository: users tell the rule broker about the services being searched. The rule broker knows or identifies (e.g. via WSDL and SOAP) appropriate service providers and installs a suitable rule at the remote provider (that must support this XML-ECA functionality), who then notifies the original user directly.

The main ideas for implementing the proposed system are presented. Of importance here is the reuse of several current Web standards and of their implementations. The DOM Event Model [dom00], an XQuery engine, and a generator of SOAP calls are needed.

Though the above approach only deals with notifications, its general architecture seems to be extensible to update actions too. The approach including the rule broker is likely to provide a good basis for Semantic Web ECA rules.

ECA-based Communication: XChange

The language **XChange** (see also Section 3.1 for a description of its update functionality and the underlying query language Xcerpt) aims at establishing reactivity, expressed by *reaction rules*, as communication paradigm on the Web. In XChange the events are not restricted to changes in a database, but an event language will be designed that covers update and message events as atomic events, as well as composite events.

The data that is communicated between Web sites are called events in XChange. An event is an XML instance with a root element with label `event` and the following four parameters (represented as child elements as they may contain complex content): `raising-time` (i.e. the time of the raising machine when the event is raised), `reception-time` (i.e. the time of the receiving machine when the event is received), `sender` (i.e. the URI of the site where the event has been raised), and `recipient` (i.e. the URI of the site where the event has been received). An `event` is an envelope for arbitrary XML content, and multiple `events` can be nested (e.g. to create trace histories).

An important distinction needs to be made between (i) static or persistent data, i.e. data of Web pages, and (ii) dynamic or volatile data, i.e. events. For querying static data (standard) queries are used, and persistent data is updatable. For querying dynamic data, *event queries* are used. Volatile data is *not* updatable. As events are XML instances, like static data, the same query language is used for querying static and dynamic data.

XChange events are directly communicated between Web sites without a centralized processing or management of events. All parties have the ability to initiate a communication. Since communication on the Web might be unreliable, synchronisation is supported by XChange.

Events are processed locally at the recipient Web site by means of XChange ECA rules. The event part is expressed by an event query, i.e. a query against events received by the Web site. Two kinds of event queries are supported: atomic event queries (i.e. one event query term) and composite event queries. Composite event queries (CEQ) are formed from atomic event queries and/or composite event queries along three dimensions: temporal range (e.g. CEQ within duration), event composition (e.g. not CEQ in finite time interval), and occurrence (specifying multiplicity, position, and repetition). The condition part is expressed by an Xcerpt query against (local or remote) Web resources. For the action part, XChange considers *transactions* instead of isolated actions. A transaction is a group of updates and/or explicit events (i.e. events that are raised and sent to other Web sites), with the A(C)I(D) properties. Atomicity (A) and isolation (I) are considered in XChange, the issue of consistency (C) and durability (D) for transactions are currently not investigated in the project.

An XChange prototype will be available in the near future.

In this approach, the events must occur locally – either as changes in the database or by messages (which, by the *push* communication strategy can also include messages about events occurring somewhere else in the Web). ECA rules are then evaluated locally, and appropriate actions (updates, or sending messages) are taken.

ECA Rules in "Closed Communities": Active Views

The **Active View** system [AAC+99] aims at providing a Web-based environment for collaborative applications. An *Active View* is not only a view, but it defines a kind of a simple agent pattern whose instances allow actors to perform controlled activities and to work interactively in a distributed environment. Central to an Active View infrastructure is a repository, and a management for active rules and communication. The specification of an active view consists of the following components:

- a view on the underlying (XML) data in a repository,
- method specifications that allow the actor to update the repository via the view,
- activities specifications: these activities can also update the view, and can be invoked declaratively from other view instances,
- active rules in the form of ECA rules.

An application of active views is illustrated in [AAC+99]: by logging on an e-commerce site, a customer invokes the customer pattern, and then has access to the repository and can communicate, e.g. with a vendor instance. The view data consists of the catalog items, equipped with methods that update the repository (e.g. to submit an order), and a dispatcher can assign the consumer (i.e., his view) to a vendor with whom he can communicate then (by calling activities of the vendor view). Active rules can both be associated with the central instance, and with views.

Reactivity. At the time this work was reported, there was no standard query language for XML; *Active Views* is based on a simple query and update language inspired by Lorel [AQM+97], including an update construct.

Events are (remote) method calls, operations like write, read, append, or detection of changes. Note that in contrast to most other approaches, events are not necessarily located at the same place as the active rules: events occurring in the central repository (invoked either by local reactive rules, or by activities of some views) are communicated to the views. Views are either notified by all events, or notification can be customized as subscription to certain events. The condition part of an *Active View* condition is an XML query that is evaluated as a boolean. The action part can specify operations on the repository, method calls (remote, but inside the Active View system) or notifications.

Further Functionality. The approach of *Active Views* goes beyond active rules, and incorporates many other issues that are relevant to the Semantic Web:

- distribution of data, events, and activities;
- update propagation to the views[2]: if the repository is updated by some activity, views must be updated or become stale. The propagation of changes from the repository to the views is supported, and alternative ways are sketched;
- communication of events (notifications, subscriptions, detecting remote events);
- access rights when using views (e.g., in the above-mentioned application, "bad" customers can be excluded by using a blacklist).

On the other hand, *Active Views* restricts reactivity to a "closed community", and to a predefined set of rules.

[2]Note that update propagation through the Web will raise further issues.

4.2.2 Event-Condition-Action Rules for the Semantic Web

Reactivity is an important component of the vision of the Semantic Web as a closely intertwined network of autonomous nodes. In contrast to the current Web, where many sites are just providing information, and others simply query them on-demand, the Semantic Web will profit from enhanced communication between its nodes, not only for answering queries, but also for its evolution. It is crucial that relevant changes to information that has been used by a Semantic Web agent are consistently and rapidly propagated to all interested parties.

The ECA Approach Proposed in [PPW03]

The **Event-Condition-Action language** proposed in [PPW03], and described in the previous section, can also be used for RDF data which has been serialized as XML data. [PPW03] reports on the first steps towards such a language for RDF data, and presents a set of examples working directly with the metadata expressed in RDF/RDFS. [PPW04] defines a query language that works directly on the RDF graph/triple representation, and extends it with the above ECA functionality. The events and conditions are as before, but now working on RDF level; the action part is now a sequence of actions, where each action represents insertion or deletion of a set of RDF triples (i.e. of the form (*subject*, *predicate*, *object*))

A distributed version supporting ECA rules on distributed RDF repositories is to be developed as part of the SeLeNe project[3]. The project investigates self e-learning networks, where such a network is a distributed repository of metadata related to learning objects.

A distributed system architecture is proposed, which offers a context for discussion on problems like registering of ECA rules and rule triggering, and execution in distributed environments. The architecture contains peers and super-peers, which coordinate a group of peers. However, the accent is on infrastructure issues and not, for example, on means for communication between the peers of the network.

XChange

XChange, as a language for specifying reactivity on the Web, has been discussed in the previous section. In [BFPS04b] it is shown that the language **XChange** can be used to specify propagation of changes on the Semantic Web. Using a simple reasoner implemented in the underlying query language Xcerpt, examples working with RDF data serialized as Xcerpt terms are presented.

Applying the languages Xcerpt and XChange to more complex Semantic Web applications is currently being investigated and may result in the implementation of (partial) reasoners for certain ontologies.

4.2.3 Active behavior encoded in XML Data

Relational databases distinguish between data (stored in relations) and activities (by procedures, functions and triggers). With object-oriented databases, this distinction has been blurred since objects encapsulate both data and behavior. But still, the behavior is local to the database.

With XML data, XML syntax of query languages (XQueryX), and XML-based Web Services (WSDL, UDDI, SOAP) the encoding of active behavior directly in the data becomes possible.

[3]SeLeNe project, `http://www.dcs.bbk.ac.uk/selene`

This approach is followed by **Active XML**[4] [AXM02, ABM+02]. In Active XML Service calls are put into XML instances by special elements of the form

<sc>*service-URL arguments*</sc>

where *arguments* contains a (possibly empty) sequence of elements that are submitted as arguments with the service call. The XML elements resulting from the service call are inserted into the document as siblings of the call.
Active XML is employed in the DBGlobe system [PAP+03], aiming at

> " ...viewing the conglomeration of interconnected peers carrying data as a virtual super-database ..."

[4] http://www-rocq.inria.fr/gemo/Gemo/Projects/axml/

Chapter 5

Rule-Based Agents

To date there exist different approaches, frameworks and platforms to develop agents on different levels of abstraction. For example, web services, the common object request broker architecture [Obj92, Obj93], and the parallel virtual machine [GBD+94] cater for the infrastructure of distributed systems, in general, and multi-agent systems, in particular. Such systems tackle low-level issues of distributed systems and form the technical basis of high-level conceptual frameworks for designing and developing agent applications. Programming languages such as April [MC95] or PVM-Prolog [CM96, CM97] support distributed computing and declarative programming. Unfortunately, on the highest level of abstraction one mostly finds either agent theories which are not operational, such as the *BDI logics* of [RG91], or implemented systems with certain data structures corresponding to beliefs and goals [IG90, LHDK94] but without a formal semantics, such as PRS, dMARS, and ARCHON. We start this Chapter, in Section 5.1, by giving an overview an Vivid agents [SW00], that sets out to bridge this gap by defining an agent framework and its web-enabled implementation. Other rule-based agent frameworks have been developed recently also with the goal of both being operational and providing a formal logical semantics. Examples of such frameworks are the DALI logic programming agent-oriented language [CT02, CT04], MetateM [BFG+95] and Concurrent MetateM [Fis93], 3APL [HdBvM98, HBdHM99] and $\mathcal{MINERVA}$ [Lei03, LAP02]. Rather than making here an extensive overview of all these systems, in Section 5.2 we concentrate on one of these approaches and make a brief overview on the agent systems being developed within the project SOCS – Societies Of ComputeeS (IST-2001-32530).

Finally, in Section 5.3, we briefly overview IMPACT [RRS96, SBD+00], an agents framework aiming at 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.

5.1 Vivid Agents

In this section we overview work done on reaction rules for rule-based Vivid agents [SW00]. An artificial agent is called *rule-based*, if its behavior and/or its knowledge is expressed by means of rules. More specifically, in this section we review a general architecture for rule-based agents originally proposed in [Wag96, SW00, DKSW03] and discuss how it can be realized with the help of Semantic Web languages.

Agents are situated in an environment and exhibit reactive behavior [WJ95]. *Reaction rules* are natural means to specify such agent behavior. Reaction rules generalize *event-condition-action rules* used in active databases [MD89, KGB+95].

5.1.1 A Basic Architecture for Rule-Based Agents

In philosophy and AI, there is a strong tradition to describe rational agents in terms of their beliefs, desires, and intentions (BDI) [Bra87, CL90]. While many researchers (see e.g. [RG91]) follow the philosophical logic tradition in modelling mental components with the help of highly complex multi-modal logics, another strand of research takes a more practical approach and models mental components as specific data structures forming the composite state of agents whose operational semantics is given by the resulting state transition system. In his seminal paper [Sho93], Shoham coins the term *agent-oriented programming* (AOP), which is centered around the three mental components of beliefs, capabilities, and commitments. In the next section, we will sketch an agent framework based on knowledge and perception.

Knowledge- and Perception-Based (KP) Agents

While we can associate *implicit* notions of *goals* and *intentions* with any "intentional system", be it natural or artificial (according to D. Dennett), it is only the *explicit* notion (of a goal or an intention) which counts for an artificial agent from the programming point of view. Having an explicit goal requires that there is some identifiable data item in the agent program which represents exactly this goal, or the corresponding sentence. Having explicit goals makes only sense for an agent, if it is capable of generating and executing plans in order to achieve its goals. Simple agents, however, which are purely *reactive*, do not generate and execute plans for achieving explicit goals assigned to them at run time (i.e. do not behave *pro-actively*), but only react to events according to their reactive behavior specification. Of course, a reaction pattern can be viewed as encoding a certain task or goal which is implicit in it. But unlike explicit goals, such implicitly encoded tasks have to be assigned to the agent at design time by hard-coding them into the agent system.

So what are the basic components shared by all important – and even very simple – types of agents? At any moment, the state of any such agent comprises *beliefs* (about the current state of affairs) and *perceptions* (of communication and environment events), and possibly other components such as tasks/goals, intentions, obligations, emotions, etc. While the agent's beliefs are represented in its *knowledge base* (KB), its perceptions are represented (in the form of incoming messages) in its *event queue* (EQ). We obtain the following picture:

$$\text{agent state} = \text{beliefs} + \text{perceptions} + \ldots$$

or, formally,

$$A = \langle KB, EQ, \ldots \rangle$$

And the state of a purely reactive agent may very well consist of just these two components, and nothing else:

$$\text{reactive agent specification} = \text{reaction patterns} + \text{initial state}$$
$$\text{reactive agent state} = \text{beliefs} + \text{perceptions}$$
$$(\text{or, formally, } A = \langle KB, EQ \rangle)$$

Example 5.1 *Consider the following example: A personal finance agent monitors a user's portfolio and acts if appropriate by alerting the user to critical situations, as well as selling and buying of shares.*
The behavior of the agent is specified by the following rules:

1. *If an investment is critical and the investment is for more than 1 year in the portfolio then sell the investment.*
2. *If an investment is critical then send an alert with high priority.*
3. *If an investment is at risk then send an alert.*
4. *If the value of the investment dropped by more than 5% then the investment is* critical.
5. *If the value of the investment dropped by more than 3% and the value of the investment in the portfolio is more than 10% of the total value of all investments then the investment is* critical.
6. *If the value of the investment dropped by more than 3% then the investment is* at risk.
7. *Log the investment. (No prerequisites)*

The above rules are a mix of derivation and reaction rules and in the remainder of the paper, we will develop and implement this example in our framework.

The core of any reactive agent is its KB. Technically, the beliefs in a KB are expressions in some representation language.

Example 5.2 *For instance, beliefs in a KB may be simple attribute/variable=value pairs like*

$$\begin{aligned} \mathit{MyName} &= 007, \text{ or} \\ \mathit{FaxNo[sunshine\ Ltd]} &= 8132, \end{aligned}$$

such as in a conventional program, or atomic sentences like

$$\begin{gathered} \mathit{I\_am(007)}, \text{ or} \\ \mathit{company(sunshine\ Ltd, malibu, 8132)}, \end{gathered}$$

such as the table rows in a relational database, or the facts in a Prolog program. In certain cases, beliefs may have to be qualified, e.g. by a degree of uncertainty, a valid-time span, or a security classification, like in

$$\begin{gathered} \mathit{price(sunshine\ Ltd, rise)} : \text{very\_likely} \\ \mathit{strategy(cautious)} \ @ \ \mathit{[2001/05/01–\infty]} \\ \mathit{price(dodgyCorp, fall)} \ / \ \text{top\_secret} \end{gathered}$$

Perceptions may have the form of typed messages labelled with their origination, such as the environment event message

$$\langle \ \mathit{observed(dog(approaching,\ 300m){:}0.7)}, \ \ \mathit{camera\_1} \ \rangle,$$

or the communication event message

$$\langle \ \mathit{tell(price(sunshine\ Ltf,\ rise),\ aFriend)} \ \rangle$$

and FIFO-buffered in the event queue EQ.

Thus, all interesting types of artificial agents are *knowledge- and perception-based (KP).*

5.1.2 Reaction Rules

Let us now consider the reaction rules of an agent. Reaction rules encode the behavior of KP agents in response to perception events created by the agent's perception subsystems, and to communication events created by communication acts of other agents. They are similar to *event-condition-action (ECA)* rules known from 'active' databases [MD89, KGB+95] (see Chapter 4. We distinguish between mental, physical, and communicative reaction rules.

Definition 5.1 *Reaction Rule*
Let A_1, A_2 be agent terms. Let L_{Evt}, L_{Com} and L_{Act} be an environment event, a communication event, and a physical action language. Then rules of the form

$$\begin{array}{rcll} \mathit{Eff} & \leftarrow & \mathsf{recvMsg}(\varepsilon(U), A_1),\ \mathit{Cond} & \textit{(Mental)} \\ \mathsf{do}(\alpha(V)),\ \mathit{Eff} & \leftarrow & \mathsf{recvMsg}(\varepsilon(U), A_1),\ \mathit{Cond} & \textit{(Physical)} \\ \mathsf{sendMsg}(\eta(V), A_2),\ \mathit{Eff} & \leftarrow & \mathsf{recvMsg}(\varepsilon(U), A_1),\ \mathit{Cond} & \textit{(Communicative)} \end{array}$$

where $\mathit{Cond} \in L_{\mathrm{Query}}$, $\mathit{Eff} \in L_{\mathrm{Input}} \cup \{\mathsf{Goal}(g),\ \mathit{not}\ \mathsf{Goal}(g) \mid\ g \in L_{\mathrm{Input}}\}$, and $\varepsilon(U) \in L_{\mathrm{Evt}} \cup L_{\mathrm{Com}}$, $\alpha(V) \in L_{\mathrm{Act}}$, $\eta(V) \in L_{\mathrm{Com}}$ are called mental, physical and communicative reaction rules.

The event condition $\mathsf{recvMsg}(\varepsilon(U), A_1)$ is a test whether the event queue of the agent contains a message of the form $\varepsilon(U)$ sent by some perception subsystem of the agent or by another agent identified by A_1, where $\varepsilon(U)$ represents an environment or a communication event, and U is a suitable list of parameters. The epistemic condition *Cond* refers to the current knowledge state, and the mental effect *Eff* specifies an update of the current knowledge state and/or the adoption or deletion of goals. For physical reactions, L_{Act} is the language of all elementary physical actions available to an agent; for an action $\alpha(V)$, $\mathsf{do}(\alpha(V))$ calls a procedure realizing the action α with parameters V. For the communicative reaction, $\mathsf{sendMsg}(\eta(V), A_2)$ sends the communication message $\eta(V)$ with parameters V to the receiver A_2.

Example 5.3 *Both perception and communication events are represented by incoming messages. In a robot, for instance, appropriate perception subsystems, operating concurrently, will continuously monitor the environment and interpret the sensory input. If they detect a relevant event pattern in the data, they report it to the knowledge system of the robot in the form of a perception event message. Similarly, the portfolio agent monitors the share prices and updates of the prices are processed as incoming messages.*

In general, reactions are based both on perception and on knowledge. Immediate reactions do not allow for deliberation. They are represented by rules with an empty epistemic premise, i.e. $\mathit{Cond} = \mathit{true}$. Timely reactions can be achieved by guaranteeing fast response times for checking the precondition of a reaction rule. This will be the case, for instance, if the precondition can be checked by simple table look-up such as in relational databases or factbases. Reaction rules are triggered by events. The agent interpreter continuously checks the event queue of the agent. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfiable in the current knowledge base, all free variables in the rules are instantiated accordingly resulting in a set of triggered actions with associated mental effects. All these actions are then executed, leading to physical actions and to sending messages to other agents, and their mental effects are assimilated into the current knowledge base.

5.2 Agent Systems Developed in SOCS

As mentioned in the introduction to this chapter, instead of making here an extensive overview on the various rule-based agent frameworks, we concentrate on, and make a brief overview of, the agent systems being developed within the project SOCS – Societies Of ComputeeS (IST-2001-32530)[1]. Comparisons of these systems to other rule-based agent frameworks can be found in the literature cited in this section.

The SOCS project, funded by the European Commission under the 5th framework, aims at providing a computational logic model for the description, analysis, and verification of global and open societies of heterogeneous computees. Computees are understood in SOCS as abstractions of the entities (or agents) that populate open and global computing environments. As with Vivid agents, the SOCS project also aims at bridging the gap between low-level approaches to agents, with no obvious logical characterization allowing for a proper analysis and verification, and more abstract specifications that have no computational counterpart. Accordingly, the models for societies of computees that SOCS aims at providing should have a computational counterpart that is executable but at the same time provably correct with respect to the formal models.

As in the REWERSE *Working Group I5 "Evolution and Reactivity"*, SOCS also aims at dealing with dynamic environments, i.e. with the possibility of evolution, and with incomplete information about the environment. But SOCS goes beyond the goals of REWERSE regarding the behavior of agents. In fact, in SOCS, computees' behavior should cater for more than reactivity, and issue like planning activities to accomplish goals, sharing of resource to achieve goals, revision of computees beliefs due to their incompleteness and possible misconception about the environment are being developed within SOCS. On the other hand, the specific requirements of being able to deal with more complex data structures and models related to the Web and the Semantic Web, as well as the capability to deal with richer event languages and transactions, to be worked out in REWERSE, are beyond the scope of SOCS.

One of the main basic choices in SOCS, and in their developed agent systems, is the use of Computational Logics and, more specifically in most cases, Abductive Logic Programming [KKT98] for the modelling and realization of agents. Abductive Logic Programming is used as a means to reconcile rational behavior of an agent, required for e.g. planning activities with reactive behavior. This reconciliation is made on the basis of an *"Observe-Think-Act"* cycle of agents that has been proposed in [KS96, KS99] and for which an abductive proof procedure has been defined [FK97] and implemented. This procedure has been recently extended to cope also with constraint solving (CIFF) [EMS$^+$04b], making it possible to deal with time intervals.

According to [KS96, KS99] knowledge is represented by a logic program augmented with integrity constraints and sets of goals. The integrity constraints can be denials, expressing prohibitions, e.g. $do(agent, Act_1, T) \wedge do(agent, Act_2, T) \wedge Act_1 \neq Act_2 \rightarrow false$ (stating that *agent* cannot perform two actions at the same time), and also rules, similar to (event-)condition-action rules. For example, the integrity constraint

$$at(Agent, B, T) \wedge intruder(Agent) \wedge at(self, a, T) \rightarrow do(self, move(A, B), [T, T + 10])$$

states that agent $self$ should move to a location where an intruder has been detected, in a moment between the time the intruder has been detected and that time plus 10 units. Goals

[1] More information on the SOCS project can be found at `http://lia.deis.unibo.it/Research/Projects/SOCS/`

are sets of predicates that the agents seeks to make true. An agent evolves according to a cycle consisting of the following activities: new observations, acquired from the environment, are added to the knowledge – *observe*; a fixed number of derivation (rewriting) steps for proving the current goals are performed by using the rules (in a backward manner) and the integrity constraints (in a forward manner) in which actions are hypothesized, or abduced – *think*; an action among those abduced is chosen for actual execution – *act*. This way, a purely reactive behavior is obtained if all integrity constraint rules only have in their bodies conditions that are either immediately obtained from the observed facts or from other facts in the knowledge base. Rationality behavior is obtained when checking the conditions requires various derivation steps. A combination of both is obtained by limiting the number of possible steps in each iteration of the cycle.

This basic approach has been extended to deal with negotiation of resources between agents [STS02]. This is done by modifying the agent cycle sketched above in order to deal with dialogues between agents and to enforce atomicity and interleaving of dialogues. Dialogues here are understood as sequences of dialogue moves, each move being an utterance (e.g. request, give, accept, refuse, etc.). This framework has been further extended in a number of ways. One such extension is related to the ability of negotiating not just resources, but also time windows during which resources are shared [STS03].

Other extensions introduce different levels of conformance to check and enforce the adaptation of an agent to public protocols regulating the interaction in a multiagent system [EMST03, EMST04]. For this, the framework is extended to deal with rules and constraints on utterances. The work on protocols and verification of compliance to protocols in agent societies is being further explored in the context of SOCS, with recent results [ACG+03, ADG+04] where *Social Integrity Constraints* and ways to express expectations of agents are defined. Social integrity constraints specify the way expectation should be generated, given a partial history of a society of agents, i.e. given a sequence of events that had occurred. In [ADG+04] it is shown how social integrity constraints may be used to implement communication protocols, including the FIPA Contract Net Protocol [FIP02c].

Implementations have accompanied most of the work described above. In particular, a platform for programming software agents (PROSOCS) [SKL+04], a society infrastructure tool (SOCS-SI) [ACT04], and an engine for the CIFF procedure [EMS+04a] have been designed and implemented.

In this section some work being developed in SOCS that may be relevant for REWERSE has been surveyed, concentrating on a few aspects of the work in SOCS, viz. dealing with rationality, beyond reactivity, and communication protocols in societies of agents. This way, this overview must be understood as giving only a partial view of SOCS. In fact more work, not even mentioned here, is being developed. For a complete account of this work, the reader is referred to the SOCS homepage, where a complete list of publications and demos of implementations may be found. Publications on that list contain various comparisons to other agent systems, such as the ones mentioned in this chapter's introduction.

5.3 IMPACT

The Interactive Maryland Platform for Agents Collaborating Together (IMPACT) [RRS96, SBD+00] is an international research project led by the University of Maryland. Its main goal 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. Since we are concerned with the interaction between several distributed Web resources — reacting to each others messages — and their knowledge evolution, the relevance of IMPACT to our work becomes immediate. In this chapter we make a brief overview of IMPACT [SBD+00] and its underlying concepts, and also take a brief glance over the use of Deontic Logics in IMPACT.

The IMPACT Team had as objectives to develop techniques and tools to build agents on top of existing legacy code — as well as on top of freshly written code — and at the same time enabling agent interoperability. We are set to investigate and design the reactivity and knowledge evolution principles of a network of several distributed Web resources, some of which may be built on top of legacy systems.

Under IMPACT's philosophy, each agent should offer some data services that must be somehow described. Different agents can possibly use different ontologies to describe their services, their environments and their behaviors. IMPACT also concerns about the organization of the thousands of networked agents who offer their services to others and to humans, and the facilitation of the collaboration between such agents.

In all, IMPACT is a powerful system and it has proved to be useful being used in practical fields such as Army applications, Logistics, and Air Traffic Control. The main features of this system's architecture, and the core agent concept and architecture are IMPACT's primary differences to other agent-based systems and are the source of IMPACT's success. Moreover, IMPACT's use of a declarative language for specifying the agent program greatly contributes to the flexibility in agent-design as well as facilitating the programming/correcting process.

5.3.1 Agent Architecture in IMPACT

As different application programs reason with different types of data and as even programs that deal with the same types of data often manipulate that data in a variety of different ways, it is critical that any notion of agenthood be applicable to arbitrary software programs. Agent developers should be able to select data structures that best suit the application functions desired by users of the application they are building.

It is important to assure that all agents have the same architecture and hence the same components, although the content of these components can be different. This may lead to a variety of different behaviors and capabilities offered by different agents.

IMPACT's agents' architecture is comprised of the following components:

Data structures: A specification of the data types or data structures that the agent manipulates. As usual, each data type has an associated domain which is the space of objects of that type. For example, the data type *countries* may be an enumerated type containing names of all countries. At any given point, the instantiation or content of a data type is some subset of the space of the dataobjects associated with that type.

The set of data structures is manipulated by a set of functions that are callable by external programs (such functions constitute the Application Programming Interface or API of the package on top of which the agent is being built. An agent includes a specification of all these API function calls' signatures (i.e. types of the inputs to such function calls and types of the output of such function calls).

Message box: 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.

Actions: 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.

Concurrency: 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, being sequentially ordered execution one of the simplest choices.

Action constraints: Each agent has a set of action constraints which are rules of the form "If the state satisfies some condition, then actions $\{a_1, \ldots, a_n\}$ cannot be concurrently executed."

Integrity constraints: Each agent has a set of integrity constraints that states of the agent are expected to satisfy. Such integrity constraints are of the form "If some condition C is true, then an atom A must be true."

Agent program: 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 don't'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 (cf. below) to implement what the agent can and cannot do. If $\alpha(\overrightarrow{t})$ is an action with parameters $\overrightarrow{t}$, then $O\alpha(\overrightarrow{t})$; $P\alpha(\overrightarrow{t})$; $F\alpha(\overrightarrow{t})$; $Do\,\alpha(\overrightarrow{t})$; $W\alpha(\overrightarrow{t})$ are called *action status atoms*. These action status atoms are read (respectively) as $\alpha(\overrightarrow{t})$; is obligatory, permitted, forbidden, to be done, and the obligation to do $W\alpha(\overrightarrow{t})$; 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

$$A \leftarrow \chi \wedge L_1 \wedge \ldots \wedge L_n$$

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

Agent behavior: We assume that when the agent is initially constructed and deployed, all integrity constraints are satisfied by the agent's state. This is analogous to requiring that the agent, when built and deployed, is not "messed up" right at the beginning. Given this, we can 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. We assume that 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. There is no loss of generality in making this assumption. Thus, every time 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, and
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. The important point to remember is that agents continuously respond to messages (state changes) by computing such a status set, and concurrently executing all the actions of the form $Do\,\alpha$ in that status set.

5.3.2 IMPACT Architecture

The IMPACT system architecture comprises several components, namely: an Agent Development Environment (AgentDE for short), the IMPACT Server, the Agent Roost, the IMPACT Connections module and the AgentLog component. Most of these components have a practical interest and provide useful functionalities to the IMPACT system, but their conceptual interest to our work seems to be limited. The Agent DE component, however, has some functionalities which may be of importance.

Besides providing an environment within which an agent developer can program all the parts of an agent, the AgentDE performs a series of validations and verifications on the agent code before the agent can be deployed.

Among these validations and verifications the AgentDE performs the "Deontic Stratification" — an operation concerning the Deontic Logic Operators which are described below.

5.3.3 Deontic Logics in IMPACT

IMPACT used Deontic Logics [Tho03] for reasoning about and working with actions.

The Deontic Logic Operators (DLOs) used in IMPACT are

O — Obligatory

P — Permitted

F — Forbidden

W — Waived

Do — Execute

These DLOs are used for programming the rules that form the agent program. An agent program rule looks like

$$A \leftarrow \chi \wedge L_1 \wedge \ldots \wedge L_n$$

where A is an action status atom, χ is a condition, and $L_1, \ldots, L_n$ are action status literals as explained in Section 5.3.1. As it is obvious, DLOs are naturally well suited for integrity constraint specification as well as for action constraint specifications.

In [Tho03], Simon Thornton shows the clear similarities between some classical Deontic Logic Operators and the classical Modal Logic '$\Box$' and '$\Diamond$' operators. Additionally, Thornton shows that an already built and used proof workbench tool — originally designed for Modal Logic — can be used to efficiently perform automated reasoning using Deontic Logic Operators. This shows that it is feasible to develop an efficient logic based system to reason with the (deontic) logic rules which make up the agent program.

Deontic stratification

The check for Deontic Stratification ensures that an action status atom is never defined recursively in terms of its own negation. For instance, a rule such as

$$O\alpha \leftarrow \neg P\alpha$$

is a rule that defines $O\alpha$ cyclically (implicitly whenever $O\alpha$ holds, $P\alpha$ must hold as well).

IMPACT deals with this kind of problem — definition by self-negation — by simply not allowing it to occur; although the detection of such *'looping-across-an-odd-number-of-negations'* definitions might be computationally expensive and time-consuming.

The Deontic Stratification, being an interesting feature, has also some drawbacks, namely, if the agent's program gets updated — either by an internal self-update or by an external-agent message which causes the update — the resulting updated program might no longer be Deontic Stratifiable. This would render the agent useless. Malicious-intentioned users could use this security weakness to do harm to a system. To prevent such an undesirable situation, one must perform the Deontic Stratification process every time an agent's program is updated; which turns out to be a very time-consuming task as the agent's program grows larger and larger.

Acknowledgements

We would like to thank Thomas Eiter and Gerd Wagner, who acted as cross readers of this deliverable, for their valuable comments on a previous version of this document, which really helped on improving it.

Bibliography

[AAC+99] Serge Abiteboul, Bernd Amann, Sophie Cluet, Adi Eyal, Laurent Mignet, and Tova Milo. Active Views for Electronic Commerce. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1999.

[ABBL04] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*, Berlin, 2004. Springer.

[ABLP02] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer-Verlag, 2002.

[ABM+02] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *Very Large Data Bases Conference (VLDB'02)*, pages 1087–1090. Morgan Kaufmann, 2002.

[ACG+03] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. In W. van der Hoek, A. Lomuscio, E. de Vink, and M. Wooldridge, editors, *Proceedings of the Workshop on Logic and Communication in Multi-Agent Systems (LCMAS)*, 2003.

[ACT96] ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *SIGMOD Record*, 25(3):40–49, September 1996.

[ACT04] M. Alberti, F. Chesani, and P. Torroni. SOCS-SI. In C. Sierra L. Sonenberg, editor, *3rd International Conference on Autonomous Agents and Multi Agent Systems*. Systems Demos, 2004.

[ADG+04] M. Alberti, D. Daolio, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and Verification of Agent Interaction Protocols in a Logic-based System. In Hisham M. Haddad, Andrea Omicini, and Roger L. Wainwright, editors, *Proceedings of the 19th ACM Symposium on Applied Computing (SAC 2004)*, pages 72–78, 2004.

[AGM85] C. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symbolic Logic*, 50(2):510–530, 1985.

[AH98] S. F. Andler and J. Hansson, editors. *Proceedings of the 2nd International Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *Lec-*

ture Notes in Computer Science. Springer, 1998. ISBN 3-540-65649-9.

[ALP+00] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, 2000. A shorter version appeared in "Principles of Knowledge Representation and Reasoning'98".

[APPP02] J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Int. Journal on Digital Libraries*, 1(1):68–88, 1997.

[AS97] J.v.d. Akker and A. Siebes. Enriching Active Databases with Agent Technology. In P. Kandzia and M. Klusch, editors, *Proceedings of the First International Workshop on Cooperative Information Agents (CIA-97)*, volume 1202 of *Lecture Notes in Artificial Intelligence*, pages 116–125. Springer, 1997.

[AVFY98] Serge Abiteboul, Victor Vianu, Bradley S. Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 179–187, 1998.

[AWH95] A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems (TODS)*, 20(1):3–41, March 1995.

[AXM02] Active XML Primer, 2002. `http://www-rocq.inria.fr/gemo/Gemo/Projects/axml/`.

[BBCC02] Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *Intl. Conference on Data Engineering (ICDE)*, pages 403–418, San Jose, California, 2002.

[BCD94] A. Buchmann, S. Chakravarthy, and K. Dittrich. Active databases. Dagstuhl Seminar No. 9412, Report No. 86, 1994.

[BCL97a] M. Berndtsson, S. Chakravarthy, and B. Lings. Extending Database Support for Coordination Among Agents. *International Journal on Cooperative Information Systems*, 6(3-4):315–339, 1997.

[BCL97b] M. Berndtsson, S. Chakravarthy, and B. Lings. Result Sharing Among Agents Using Reactive Rules. In *Proceedings of the First International Workshop on Cooperative Information Agents (CIA-97)*, volume 1202 of *Lecture Notes in Artificial Intelligence*, pages 126–137. Springer, February 1997.

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BCP00] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active Rules for XML: A New Paradigm for E-Services. In *First Workshop on Technologies for E-Services (TES 2000)*, September 2000.

[BCP01] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *World Wide Web Conf. (WWW 2001)*, pages 633–641, 2001.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrandt, and D. Suciu. A query language and

optimization techniques for unstructured data. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 505–516, Montreal, Canada, 1996.

[Ber90] Brian Berliner. CVS II: Parallelizing Software Development. In USENIX Association, editor, *Proc. of the Winter 1990 USENIX Conference*, pages 341–352, Washington, DC, USA, January 1990.

[BFG+95] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: An introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.

[BFG01] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 119–128, 2001.

[BFL99] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, pages 79–93, Cambridge, November 1999. MIT Press.

[BFPS04a] François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data retrieval and evolution on the (semantic) web: A deductive approach. Technical Report PMS-FB-2004-13, University of Munich, May 2004.

[BFPS04b] François Bry, Tim Furche, Paula Lavinia Pătrânjan, and Sebastian Schaffert. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In *Proc. of Workshop on Principles and Practice of Semantic Web Reasoning, St. Malo, France, (6th – 10th September 2004)*, 2004.

[BGK+95] J. A. Bailey, M. Georgeff, D. B. Kemp, D. Kinny, and K. Ramamohanarao. Active Databases and Agent Systems - A Comparison. In T. Sellis, editor, *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 1995.

[BGK+02] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for Peer-to-Peer Computing: A Vision. In *5th International Workshop on the web and Databases (WebDB'02)*. ITC-IRST Technical Report 0204-15, 2002.

[BGP97] C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31(1–3):201–243, April–June 1997.

[BH96] M. Berndtsson and J. Hansson. Workshop Report: The First International Workshop on Active and Real-Time Database Systems (ARTDB-95). *SIGMOD Record*, 25(1):64–66, 1996.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.

[BK94] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[BK95] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *ICDT'95: Advances in Logic-Based Languages*, 1995.

[BKLW99] G.v. Bültzingsloewen, A. Koschel, P. C. Lockemann, and H-D. Walter. ECA Funtionality in a Distributed Environment. In N. W. Paton, editor, *Active Rules in Database Systems*, Monographs in Computer Science, chapter 8, pages 147–175.

Springer, 1999.

[BL92] M. Berndtsson and B. Lings. On Developing Reactive Object-Oriented Databases. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):31–34, December 1992.

[BMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *8th Annual ACM Symp. on Principles of Programming Languages*, 1981.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[BPS04a] François Bry, Paula Lavinia Pătrânjan, and Sebastian Schaffert. Poster Presentation: Xcerpt and XChange - Logic Programming Languages for Querying and Evolution on the Web. In *Proc. of 19th Int. Conf. on Logic Programming, St. Malo, France, (6th – 10th September 2004)*, LNCS, 2004.

[BPS04b] François Bry, Paula Lavinia Pătrânjan, and Sebastian Schaffert. Xcerpt and XChange: Deductive Languages for Data Retrieval and Evolution on the Web. In *Proc. of Workshop on Semantic Web Services and Dynamic Networks, Ulm, Germany, (22nd – 24th September 2004)*. GI, 2004.

[BPW02] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for xml. In *Int. WWW Conference*, 2002.

[Bra87] M. E. Bratmann. *Intentions, Plans, and practical reason*. Harvard University Press, 1987.

[BS02] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Intl. Conf. on Logic Programming (ICLP)*, number 2401 in LNCS, pages 255–270, 2002.

[BS03] A. Borgida and L Serafini. Distributed description logics: Assimilating information from peer sources. *Journal of Data Semantics*, 1:153–184, 2003.

[BS04] François Bry and Sebastian Schaffert. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. of Extreme Markup Languages 2004, Montreal, Quebec, Canada, (2nd – 6th August 2004)*, 2004.

[BZBW95] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

[CAMM94] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.

[CCS94] C. Collet, T. Coupaye, and H. Svensen. NAOS - Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 132–143, 1994.

[CDGL+04] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. What to ask to a peer: Ontology-based query reformulation. In *Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 469–478, 2004.

[CDSS99] Sophie Cluet, Claude Delobel, Jérome Siméon, and Katarzyna Smaga. Your me-

diators need data conversion. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 177–188, 1999.

[CdTW00] Jianjun Chen, David J. deWitt, Feng Tian, and Yuang Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of the IBM Workshop on Logics of Programs*, number 131 in Lecture Notes in Computer Science, 1981.

[CF97] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules.* Addison-Wesley, 1997.

[CFI+00] M.J̃. Carey, D. Florescu, Z.G̃. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPeranto: Publishing object-relational data as XML. In *WebDB 2000*, 2000.

[Cha92] S. Chakravarthy, editor. *Special Issue on Active Databases*, volume 15(1–4). IEEE Quarterly Bulletin on Data Engineering, December 1992.

[Cho95a] Jan Chomicki. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming*, 25(1):1–31, October 1995.

[Cho95b] Jan Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.

[Cho95c] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[Chr93] S. Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, Edinburgh University, 1993.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB*, pages 606–617, 1994.

[CL90] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[CM96] José C. Cunha and Rui F. P. Marques. PVM-Prolog: A prolog interface to PVM. In *Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96*, Miskolc, Hungary, 1996.

[CM97] José C. Cunha and Rui F. P. Marques. Distributed algorithm development with PVM-Prolog. In *5th Euromicro Workshop on Parallel and Distributed Processing*, London, UK, 1997. IEEE Computer Society Press.

[CT02] S. Constantini and A. Tocchio. A Logic Programming Language for Multi-Agent Systems. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Logics in Artificial Intelligence, Proc of the 8th European Conference JELIA'02*, volume 2424 of *LNAI*, pages 1–13. Springer, 2002.

[CT04] S. Constantini and A. Tocchio. The DALI logic programming agent-oriented

language. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence, Proc of the 9th European Conference JELIA'04*, volume 3229 of *LNAI*, pages 679–682. Springer, 2004.

[CW96] S. Chakravarthy and J. Widom, editors. *Special Issue on the Active Database Systems*, volume 7(2). Journal of Intelligent Information Systems (JIIS), October 1996.

[Day95] U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? In *Proceedings of the 1st International Workshop on Active and Real-Time Database Systems*, Workshops in Computing, pages 3–22. Springer, 1995.

[DDDS99] M. Dekhtyar, A. Dikovsky, S. Dudakov, and N. Spyratos. Monotone expansions of updates in logical databases. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. `http://www.w3.org/TR/1998/NOTE-xml-ql`, 1998.

[DG96] Klaus R. Dittrich and Stella Gatziu. *Aktive Datenbanksysteme, Konzepte und Mechanismen*. Internat. Thompson Publ., 1996.

[Dia99] O. Diaz. Tool Support. In N. W. Paton, editor, *Active Rules in Database Systems*, Monographs in Computer Science, chapter 7, pages 127–145. Springer, 1999.

[DJ97] O. Diaz and A. Jaime. EXACT: an EXtensible approach to ACTive object-oriented databases. *VLDB Journal*, 6(4):282–295, 1997.

[DKSW03] Jens Dietrich, Alexander Kozlenkov, Michael Schroeder, and Gerd Wagner. Rule-based agents for the semantic web. *Journal on Electronic Commerce Research Applications*, 2(4):323–38, 2003.

[dom00] World Wide Web Consortium, http://www.w3.org/TR/DOM-Level-2-Events/. *Document Object Model (DOM) Level 2 Events Specification*, November 2000.

[DP97] A. Darwiche and J. Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 89(1-2):1–29, 1997.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 317–326, 1991.

[EFL+04] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.

[EFPP04] Thomas Eiter, Wolfgang Faber, Gerald Pfeifer, and Axel Polleres. Declarative planning and knowledge representation in an action language. In Ioannis Vlahavas and Dimitris Vrakas, editors, *Intelligent Techniques for Planning*. Idea Group, Inc., 2004. To appear.

[EFST01] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654, San Francisco, CA, 2001. Morgan Kaufmann Publishers, Inc.

[EFST02a] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics

based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, November 2002.

[EFST02b] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. Reasoning about evolving nonmonotonic knowledge bases. Technical Report INFSYS RR-1843-02-11, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, September 2002. *ACM Transactions on Computational Logic*, to appear.

[EFST03] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. Declarative update policies for nonmonotonic knowledge bases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*, chapter 3, pages 85–129. Springer-Verlag, 2003.

[EH82] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the 14th Annual ACM Symposium on Computing*, pages 169–180, 1982.

[EH83] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching time versus linear time in temporal logic. In *10th Annual ACM Symp. on Principles of Programming Languages*, 1983.

[EL85] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *12th Annual ACM Symp. on Principles of Programming Languages*, 1985.

[EM01] Andrew Eisenberg and Jim Melton. SQL/XML and the SQLX informal group of companies. *SIGMOD Record*, 30(3):105–108, 2001. See also `www.sqlx.org`.

[Eme90] E. A. Emerson. Temporal and modal logic. In v. Leeuwen [vL90], chapter 16, pages 995–1073.

[EMS+04a] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive Logic Programming with CIFF: System description. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence, Proc of the 9th European Conference JELIA'04*, volume 3229 of *LNAI*, pages 675–678. Springer, 2004.

[EMS+04b] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence, Proc of the 9th European Conference JELIA'04*, volume 3229 of *LNAI*, pages 31–43. Springer, 2004.

[EMST03] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 679–684. Morgan Kaufmann, 2003.

[EMST04] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Logic-based agent communication protocols. In F. Dignum, editor, *Advances in Agent Communication*, volume 2922 of *LNAI*, pages 91–107. Springer, 2004.

[FFK+97] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. STRUDEL: A web-site management system. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 549–552, 1997.

[FIP02a] FIPA ACL Message Structure Specification. Technical Report SC00061G, Foundation for Intelligent Physical Agents, Dec. 2002.

[FIP02b] FIPA Communicative Act Library Specification. Technical Report SC00037J, Foundation for Intelligent Physical Agents, Dec. 2002.

[FIP02c] FIPA Contract Net Interaction Protocol. Technical Report SC00029H, Foundation for Intelligent Physical Agents, Dec. 2002.

[FIP02d] FIPA SL Content Language Specification. Technical Report SC00008I, Foundation for Intelligent Physical Agents, Dec. 2002.

[Fis93] M. Fisher. Concurrent METATEM — A language for modelling reactive systems. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of PARLE '93 – Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 185–196, Munich, Germany, June 14–17, 1993. Springer-Verlag.

[FK97] T. Fung and R. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(2-3):189–208, 1971.

[Gab89] Dov Gabbay. The declarative past, and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in LNCS, pages 409–448. Springer, 1989.

[GB97] A. Geppert and M. Berndtsson, editors. *Proceedings of the 3rd International Workshop on Rules in Database Systems*, volume 1312 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. ISBN 3-540-63516-5.

[GB98] A. Geppert and M. Berndtsson. Workshop Report: The Third International Workshop on Rules in Database Systems (RIDS'97). *Knowledge Engineering Review*, 13(2):195–200, June 1998.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[GBLR98] A. Geppert, M. Berndtsson, D. Lieuwen, and C. Roncancio. Performance Evaluation of Object-Oriented Active Database Management Systems Using the BEAST Benchmark. *Theory and Practice of Object Systems (TAPOS)*, 4(3):135–149, August 1998.

[GD94] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases Using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering - Active Database Systems*, pages 2–9, 1994.

[GJ92] N. H. Gehani and H. V. Jagadish. Active Database Facilities in Ode. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):19–22, 1992.

[GKL97] E. Giunchiglia, G. Kartha, and V. Lifschitz. Representing actions: Indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[GKvBF98] S. Gatziu, A. Koschel, G. von Bültzingsloewen, and H. Fritschi. Unbundling Active Functionality. *SIGMOD Record*, 27(1), March 1998.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

[GL93] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[GL98a] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.

[GL98b] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.

[GLL+97] E. Giunchiglia, J. Lee, V. Lifschitz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[GLL+04] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *WebDB 1999*, pages 25–30, 1999.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[GRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Hal03] Alon Y. Halevy. Data integration: A status report. In *Datenbanken in Büro, Technik und Wissenschaft (BTW-2003)*, pages 1171–1187, 2003.

[Han96] E. N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineerin(TKDE)*, 8(1):157–172, February 1996.

[Har79] D. Harel. *First-Order Dynamic Logic*. Number 68 in LNCS. Springer, 1979.

[Har84] D. Harel. Dynamic Logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Volume II - Extensions of Classical Logic*, pages 497–604. Reidel Publishing Company, 1984.

[HBdHM99] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, November 1999.

[HdBvM98] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Formal semantics for an abstract agent programming language. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97*, volume 1365 of *LNAI*,

pages 215–230, Berlin, July 24–26 1998. Springer.

[HIST03] A. Halevy, Z. Ives, D. Suciu, and I Tatarinov. Schema mediation in peer data management systems. In *19th IEEE International Conference on Data Engineering (ICDE'03)* , pages 505–516, 2003.

[HKP82] D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, 1982.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HP00] Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed XML processing language. In *WebDB 2000*, pages 111–116, 2000.

[IBM] IBM. The Amit home page: http://www.haifa.il.ibm.com/projects/software/amit/index.html.

[IG90] François Félix Ingrand and Michael P. Georgeff. Managing deliberation and reasoning in real-time AI systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, pages 284–291, San Diego, CA, 1990.

[IS95] K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *IJCAI'95*, pages 204–210. Morgan Kaufmann, 1995.

[IS03] K. Inoue and C. Sakama. An abductive framework for computing knowledge base updates. *Theory and Practice of Logic Programming*, 3(6):671–713, 2003.

[JRB99] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Ker95] M. L. Kersten. An Active Component for a Parallel Database Kernel. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 1995.

[KFKO02] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanoff. OntoView: Comparing and Versioning Ontologies. In *Collected Posters of First Int. Semantic Web Conf. (ISWC 2002)*, Sardinia, Italy, 2002.

[KGB+95] David Kinny, Michael Georgeff, James Bailey, David B. Kemp, and Kotagiri Ramamohanarao. Active databases and agent systems – a comparison. In *Proceedings of RIDS95, International Workshop of Rules in Database Systems*, Athens, Greece, 1995.

[KK00] Kevin Kline and Daniel Kline. *SQL in a Nutshell*. O'Reilly & Associates, December 2000.

[KKT98] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logics in AI and Logic Programming 5*, pages 235–324. Oxford University Press, 1998.

[KL89] Michael Kifer and Georg Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 134–146, 1989.

[KL98] A. Koschel and Peter C. Lockemann. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Journal of Data and Knowledge Engineering (DKE)*, 25, 1998. Special Issue for the 25th Vol. of DKE.

[Kle01] M. Klein. Combining and relating ontologies: an analysis of problems and solutions. In *Proceedings of the IJCAI'01 Workshop on Ontologies and Information*

Sharing, 2001.

[KLS92] Michael Kramer, Georg Lausen, and Gunter Saake. Updates in a rule-based language for objects. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 251–262, Vancouver, 1992.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[KM91] H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR'91*. Morgan Kaufmann, 1991.

[KN03] Michel Klein and Natasha F. Noy. A Component-Based Framework for Ontology Evolution. In *Proc. of the Workshop on Ontologies and Distributed Systems (IJCAI'03)*, Acapulco, Mexico, 2003.

[KR98] G. Kappel and W. Retschitzegger. The TriGS Active Object-Oriented Database System - An Overview. *SIGMOD Record*, 27(3):36–41, September 1998.

[KS96] R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C Zaniolo, editors, *Proceedings of LID-96*, volume 1154 of *LNAI*, pages 137–149, 1996.

[KS99] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25:391–419, 1999.

[KW85] A. Keller and M. Winslett Wilkins. On the use of an extended relational model to handle changing incomplete information. *IEEE Trans. on Software Engineering*, 11(7):620–633, 1985.

[Lam80] L. Lamport. 'Sometimes' is sometimes Not Never''. In *7th Annual ACM Symp. on Principles of Programming Languages*, 1980.

[LAP02] J. A. Leite, J. J. Alferes, and L. M. Pereira. $\mathcal{MINERVA}$ - A Dynamic Logic Programming Agent Architecture. In J. J. Meyer and M. Tambe, editors, *Intelligent Agents VIII — Agent Theories, Architectures, and Languages*, volume 2333 of *LNAI*, pages 141–157. Springer-Verlag, 2002.

[LBS99] J. Lobo, R. Bhatia, and S.Naqvi. A policy description language. In *National Conference on Artificial Intelligence (AAAI)*, 1999.

[Leh01] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language (diploma thesis), August 2001. Technische Universität Darmstadt.

[Lei03] J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[Len02] Maurizio Lenzerini. Data integration: a theoretical perspective. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 233–246, 2002.

[Len03] Maurizio Lenzerini. Tutorial on information integration. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[LHDK94] Jaeho Lee, Marcus J. Huber, Edmund H. Durfee, and Patrick G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *CIRFSS94, Conference on Intelligent Robotics in Field, Factory, Service and Space*, pages 842–849. MIT Press, 1994.

[LHL+98] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with Florid: A deductive object-oriented perspective. *Information Systems*, 23(8):589–612, 1998.

[LLM98] Georg Lausen, Bertram Ludäscher, and Wolfgang May. On active deductive databases: The statelog approach. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, number 1472 in LNCS. Springer, 1998.

[LLW03] Mengchi Liu, Li Lu, and Guoren Wang. A Declarative XML-RL Update Language. In *Proc. Int. Conf. on Conceptual Modeling (ER 2003)*, number 2813 in LNCS 2813, pages 506–519, Chicago, Illinois, USA, October 2003. Springer-Verlag.

[LP98] J. A. Leite and L. M. Pereira. Iterated logic program updates. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 265–278, Cambridge, 1998. MIT Press.

[LRL+97] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–83, 1997.

[LW92] V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.

[LW00] A. Levy and D. Weld. Intelligent Internet Systems. *Artificial Intelligence*, 118(1–2):1–14, 2000.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[May01a] Wolfgang May. *A Logic-Based Approach to XML Data Integration*. Habilitation thesis, Universität Freiburg, 2001. Available at `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[May01b] Wolfgang May. A rule-based querying and updating language for XML. In *Workshop on Databases and Programming Languages (DBPL 2001)*, number 2397 in LNCS, pages 165–181, 2001.

[May01c] Wolfgang May. XPath-Logic and XPathLog: A logic-based approach for declarative XML data manipulation. Technical report, Habilitation Thesis, Universität Freiburg, Institut für Informatik, 2001. Available at `http://dbis.informatik.uni-goettingen.de/lopix/`.

[May02] Wolfgang May. Querying linked XML document networks in the web. In *11th. WWW Conference*, 2002. Available at `http://www2002.org/CDROM/alternate/166/`.

[May04] Wolfgang May. XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming*, 4(3), 2004.

[MB01] Wolfgang May and Erik Behrends. On an XML Data Model for Data Integration. In *Intl. Workshop on Foundations of Models and Languages for Data and Objects*, Viterbo, Italy, September 2001.

[MC95] F. G. McCabe and K. L. Clark. `APRIL` - Agent PRocess Interaction Language.

In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I.* LNAI 890, Springer–Verlag, 1995.

[McC90] John McCarthy. *Formalizing Common Sense.* Ablex, Norwood, 1990.

[McM93] Ken McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD-89*, pages 215–224, 1989.

[MH69] John McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4, 1969.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil90] R. Milner. *Operational and Algebraic Semantics of Concurrent Processes*, chapter 19, pages 1201–1242. Volume B: Formal Models and Semantics of v. Leeuwen [vL90], 1990.

[ML04] Wolfgang May and Georg Lausen. A uniform framework for integration of information from the web. *Information Systems*, 29(1):59–91, 2004.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1(100):1–77, 1992.

[MSL97] W. May, C. Schlepphorst, and G. Lausen. Integrating dynamic aspects into deductive object-oriented databases. In *Rules in Database Systems*, number 1312 in Lecture Notes in Computer Science, pages 20–34. Springer, 1997.

[MT94] V. W. Marek and M. Truszczyński. Revision specifications by means of programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *Proceedings of the European Workshop on Logics in Artificial Intelligence (JELIA-94)*, volume 838 of *LNAI*, pages 122–136, Berlin, September 1994. Springer.

[MZ97] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 440–451, Tucson, Arizona, 1997.

[Obj92] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Wiley, 1992.

[Obj93] Object Management Group. *Object Management Architecture Guide.* Wiley, 1993.

[OWL03] OWL-S: Web Service Ontology. `http://www.daml.org/services/owl-s/`, 2003.

[owl04] World Wide Web Consortium, http://www.w3.org/TR/owl-features/. *OWL Web Ontology Language*, February 2004.

[PAP+03] Evaggelia Pitoura, Serge Abiteboul, Dieter Pfoser, George Samaras, and Michalis Vazirgiannis. DBGlobe: a Service-Oriented P2P System for Global Computing. *SIGMOD Record*, 32(3):77–82, 2003.

[Pat99] N. W. Paton, editor. *Active Rules in Database Systems.* Monographs in Computer Science. Springer, 1999. ISBN 0-387-98529-8.

[PD99] N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[Ped89] E. Pednault. Exploring the middle ground between STRIPS and the Situation Calculus. In *Proc. of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers Inc., 1989.

[Plo81] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PPW03] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-condition-action rule languages for the semantic web. In *Workshop on Semantic Web and Databases (SWDB'03)*, 2003.

[PPW04] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Rule Languages for RDF. In *Hellenic Data Management Symposium (HDMS'04)*, 2004.

[Pra76] V. R. Pratt. Semantical considerations on Floyd-Hoare Logic. In *17.th IEEE Symp. on Foundations of Computer Science*, pages 109–121, 1976.

[Pra90] V. R. Pratt. Action logic and pure induction. In J. v. Eijck, editor, *Logics in AI: Europ. Workshop JELIA '90*, number 478 in Lecture Notes in Artificial Intelligence, pages 97–120, 1990.

[Prz88] T. Przymusinski. Perfect model semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1081–1096, Seatle, 1988. ALP, IEEE, The MIT Press.

[PS96] S. Potamianos and M. Stonebraker. The POSTGRES Rules System. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 43–61. Morgan Kaufmann, 1996.

[PT95] T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczyński, editors, *LPNMR'95*, volume 928 of *LNAI*, pages 156–174. Springer-Verlag, 1995.

[PV95] Philippe Picouet and Victor Vianu. Semantics and expressiveness issues in active databases. In *ACM Symposium on Principles of Database Systems (PODS)*, 1995.

[PW94] N. W. Paton and M. W. Williams, editors. *Proceedings of the 1st International Workshop on Rules in Database Systems*, Workshops in Computing. Springer-Verlag, 1994. ISBN 3-540-19846-6.

[RB01] E. Rahm and P. Berstein. A survey of approaches of automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[Rei93] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR91, International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1991.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RRS96] T. J. Rogers, R. Ross, and V. S. Subrahmanian. Impact: A system for building agent applications. *Journal of Intelligent Information Systems*, pages 275–294, 1996.

[RSS02] Mathieu Roger, Ana Simonet, and Michel Simonet. Towards Updates in Description Logics. In *Proc. Int. Workshop on Description Logics (DL2002)*, volume 53. Ian Horrocks and Sergio Tessaris, editors, April 2002.

[rul] ruleCore. The ruleCore home page: http://www.rulecore.com/.

[saf04] *Proceedings of the 1st International Workshop on Safety and Security in Multi-Agent Systems*, 2004.

[San94] E. Sandewall. *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

[SB01] T. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, 2001.

[SBD+00] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents*. MIT-Press, 2000.

[Sch96] S. Schwiderski. *Monitoring the behaviour of distributed systems*. PhD thesis, University of Cambridge, April 1996.

[Sel95] T. Sellis, editor. *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*. Springer, 1995. ISBN 3-540-60365-4.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[SI99] C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *LNAI*, pages 147–161, Berlin, 1999. Springer.

[Sin95] Munindar P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Intl. Workshop on Database Programming Languages*, electronic Workshops in Computing, Gubbio, Italy, 1995. Springer.

[Sin96] M. P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proc. 12th. ICDE*, 1996.

[Sin03] Thomas Sindt. Formal Operations for Ontology Evolution. In *Proc. Int. Conf. on Emerging Technologies (ICET'03)*, Minneapolis, Minnesota (USA), August 2003.

[SKD95] E. Simon and A. Kotz-Dittrich. Promises and Realities of Active Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 642–653, 1995.

[SKL+04] K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In Robert Trappl, editor, *Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4)*, pages 523–528, Vienna, Austria, 2004. Austrian Society for Cybernetic Studies.

[soa00] World Wide Web Consortium, http://www.w3.org/TR/soap. *Simple Object Ac-*

cess Protocol (SOAP) 1.1, May 2000.

[Spi00] Marc Spielmann. Verification of relational transducers for electronic commerce. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 92–103, 2000.

[SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, February 1991.

[Sti89] C. Stirling. Temporal logics for CCS. In *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, number 354 in Lecture Notes in Computer Science, pages 660–672. Springer, 1989.

[Sti95] C. Stirling. Modal and temporal logics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 477–563. Oxford Science Publications, 1995.

[STS02] F. Sadri, F. Toni, and F. Sadri. An abductive logic programming architecture for negotiating agents. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 419–431. Springer-Verlag, 2002.

[STS03] F. Sadri, F. Toni, and F. Sadri. Minimally intrusive negotiating agents for resource sharing. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 796–804. Morgan Kaufmann, 2003.

[SW95] A. Prasad Sistla and Ouri Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD 1995)*, pages 269–280, 1995.

[SW00] Michael Schroeder and Gerd Wagner. Vivid agents: Theory, architecture, and applications. *the International Journal for Applied Artificial Intelligence*, 14(7):645–76, August 2000. Francis and Taylor.

[Tho03] Simon Thornton. Automated deduction for deontic logics: Laying the foundations of legal expert systems. In *Proceedings of the First Australian Undergraduate Students' Computing Conference*, pages 100–105. Dept. of Computer Science, The Australian National University, 2003.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Halevy, and Daniel Weld. Updating XML. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 133–154, 2001.

[TRP+04] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable xml publish/subscribe system using relational database systems. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 2004.

[Uni02] Universal Description, Discovery and Integration, http://www.uddi.org. *UDDI Technical White Paper*, 2002.

[vBB95] J. van Benthem and J. Bergstra. Logic of transition systems. *Journal of Logic, Language, and Information*, 3:247–283, 1995.

[vL90] J. v. Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, 1990.

[Wag96] Gerd Wagner. A logical and operational model of scalable knowledge-and

perception-based agents. In *Proceedings of MAAMAW96, LNAI 1038*. Springer-Verlag, 1996.

[WC94] J. Widom and S. Chakravarthy, editors. *Proceedings of the 4th International Workshop on Research Issues in Data Engineering - Active Database Systems*. IEEE-CS, February 1994. ISBN 0-8186-5360-4.

[WC96a] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996. ISBN 1-55860-304-2.

[WC96b] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[WJ95] M.J. Wooldridge and N.R. Jennings. Agent theories, architectures and languages: A survey. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I*. LNAI 890, Springer–Verlag, 1995.

[wsd01] World Wide Web Consortium, http://www.w3.org/TR/wsdl/. *Web Services Description Language (WSDL)*, March 2001.

[XML00] XML:DB Initiative, http://xmldb-org.sourceforge.net/. *XUpdate - XML Update Language*, September 2000.

[xpa99] World Wide Web Consortium, http://www.w3.org/TR/xpath. *XML Path Language (XPath)*, Nov 1999.

[xqu01] World Wide Web Consortium, http://www.w3.org/TR/xquery/. *XQuery: A Query Language for XML*, Feb 2001.

[xsl99] World Wide Web Consortium, http://www.w3.org/TR/xslt/. *XSL Transformations (XSLT)*, November 1999.

[xsl01] World Wide Web Consortium, http://www.w3.org/TR/xsl/. *Extensible Stylesheet Language (XSL)*, October 2001.

[YC99] S. Yang and S. Chakravarthy. Formal Semantics of Composite Events for Distributed Environment. In *Proceedings of the 15th International Conference on Data Engineering*, pages 400–407. IEEE Computer Society Press, 1999.

[Zan94] Carlo Zaniolo. A unified semantics for active and deductive databases. In Paton and Williams [PW94], pages 271–287. ISBN 3-540-19846-6.

[ZF98] Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 403–407, Chichester, 1998. John Wiley & Sons.