REWERSE©
reasoning on the web

I5-D4

Specification of a Model, Language, and Architecture for Reactivity and Evolution

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Lisbon/I5-D4/D/PU/a1
Responsible editors:	José Júlio Alferes and Wolfgang May
Reviewers:	Mikael Berndtsson and Tim Furche
Contributing participants:	Dresden, Goettingen, Lisbon, Munich, Skoevde
Contributing workpackages:	I5
Contractual date of deliverable:	31 August 2005
Actual submission date:	31 August 2005

Abstract

After a brief introduction, this reports presents XChange, a declarative high-level language for programming reactive behavior, evolution, and distributed applications on the Web. This language description is accompanied by the corresponding declarative and operational semantics, and is illustrated in a selection of use-cases from our previous deliverable. The report then proceeds by proposing a general framework and architecture for dealing with the challenges posed by evolution and reactivity in the Semantic Web.

The report also includes two appendices with relevant work done in the project. One with a description of the reactive system ruleCore, which was developed in collaboration with industry, including the illustration of the system with the above mentioned use cases. The other appendix contains a description of the language Prova with particular emphasis on its reactive aspects, and is exemplified by implementing a use-case from the above mentioned deliverable.

Keyword List

ECA rules, Reactivity, Evolution and updates of data, Language and data heterogeneity

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

© *REWERSE 2005.*

Specification of a Model, Language, and Architecture for Reactivity and Evolution

José Júlio Alferes[1], Ricardo Amador[1], Erik Behrends[2], Mikael Berndtsson[3], François Bry[4], Gihan Dawelbait[5], Andreas Doms[5], Michael Eckert[4], Oliver Fritzen[2], Wolfgang May[2], Paula Lavinia Pătrânjan[4], Loic Royer[5], Franz Schenk[2] and Michael Schroeder[5]

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

[2] Institut für Informatik, Universität Göttingen

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

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

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

31 August 2005

Abstract
After a brief introduction, this reports presents XChange, a declarative high-level language for programming reactive behavior, evolution, and distributed applications on the Web. This language description is accompanied by the corresponding declarative and operational semantics, and is illustrated in a selection of use-cases from our previous deliverable. The report then proceeds by proposing a general framework and architecture for dealing with the challenges posed by evolution and reactivity in the Semantic Web.

The report also includes two appendices with relevant work done in the project. One with a description of the reactive system ruleCore, which was developed in collaboration with industry, including the illustration of the system with the above mentioned use cases. The other appendix contains a description of the language Prova with particular emphasis on its reactive aspects, and is exemplified by implementing a use-case from the above mentioned deliverable.

Keyword List
ECA rules, Reactivity, Evolution and updates of data, Language and data heterogeneity

Contents

1 Introduction **1**

2 The Reactive Web Language XChange **5**

2.1 Paradigms of XChange . . . 6
2.1.1 Event vs. Event Query . . . 6
2.1.2 Volatile vs. Persistent Data . . . 6
2.1.3 Rule-Based Language . . . 7
2.1.4 Pattern-Based Approach . . . 7
2.1.5 Transactional Reactivity . . . 7
2.1.6 Communication Paradigms . . . 8
2.1.7 Composite Events Defined through Event Queries . . . 8
2.1.8 Processing of Events . . . 9
2.1.9 Relationship Between Reactive and Query Languages . . . 9
2.1.10 Language Syntax . . . 10
2.2 XChange Events and Event Queries . . . 11
2.2.1 Events . . . 11
2.2.1.1 Atomic Events . . . 12
2.2.1.2 Composite Events . . . 12
2.2.1.3 Events' Occurrence Time . . . 13
2.2.2 Event Messages . . . 13
2.2.2.1 Parameters of Event Messages . . . 15
2.2.2.2 Representation of Implicit Events . . . 18
2.2.3 Event Queries . . . 19
2.2.3.1 Essential Traits . . . 19
2.2.3.2 Atomic Event Queries . . . 20
2.2.3.3 Composite Event Queries . . . 24
2.2.3.4 Legal Event Queries . . . 38
2.2.3.5 Answers to Event Queries . . . 40
2.2.4 Declarative Semantics for Event Queries . . . 43
2.2.4.1 Atomic Events and Event Stream . . . 44
2.2.4.2 Answers to Event Queries . . . 45
2.2.4.3 Relating Queries and Answers . . . 47
2.2.5 Operational Semantics: Incremental Evaluation . . . 53
2.2.5.1 Requirements and Considerations . . . 53
2.2.5.2 Evaluation of Atomic Event Queries . . . 55

2.2.5.3 Evaluation of Composite Event Queries 55
2.2.5.4 Ideas for Optimizations. 64
2.3 XChange Conditions – Web Queries 68
2.3.1 Web Queries with Xcerpt 68
2.3.2 Semantics of Web Queries: Underlying Ideas 69
2.3.3 Evaluation of Web Queries: Basic Ideas 71
2.4 XChange Actions 72
2.4.1 Update Patterns 72
2.4.1.1 Update Terms 73
2.4.1.2 Insertion Specification 74
2.4.1.3 Deletion Specification 79
2.4.1.4 Replace Specification 81
2.4.1.5 Special Case – Updating the Root 85
2.4.2 Complex Updates as Transactions 88
2.4.2.1 Elementary Updates 88
2.4.2.2 Complex Updates 89
2.4.2.3 Transactions 93
2.4.3 Semantics of Updates 95
2.4.4 Execution of Updates 97
2.5 XChange Rules 101
2.5.1 Rules 101
2.5.2 Event-Raising Rules 102
2.5.3 Transaction Rules 104
2.5.4 Deductive Rules 106
2.5.5 Range Restriction 107
2.5.5.1 Polarity of Event Queries 109
2.5.5.2 Polarity of Web Queries 110
2.5.5.3 Polarity of Actions 110
2.5.5.4 Range Restriction of XChange Rules 112
2.6 Implementation of Use Cases with XChange 114
2.6.1 Changing and Propagating Contact Information 115
2.6.2 Change of Role for a Coordinator or Deputy Coordinator 117
2.6.3 Progress reports 119
2.6.4 Late progress reports 121
2.6.5 Polls 121

3 A General Framework for Evolution and Reactivity in the Semantic Web 123
3.1 Requirements Analysis 125
3.1.1 Web vs. Semantic Web 126
3.1.2 Abstraction Levels 126
3.1.3 Domain Ontologies including Dynamic Aspects 128
3.1.4 Events 129
3.1.5 Types of Rules 133
3.2 Simple ECA Rules: Data Model Triggers 137
3.2.1 Triggers on XML Data 137
3.2.2 Triggers on RDF Data 138
3.2.3 Triggers vs. ECA Rules 140

3.3 ECA Language Structure . . . 140
3.3.1 Language Heterogeneity and Structure: Rules, Rule Components and Languages . . . 140
3.3.2 Components and Languages of ECA Rules . . . 142
3.3.3 Opaque Rules . . . 144
3.3.4 Hierarchical Structure of Languages . . . 146
3.3.5 Common Structure and Aspects of E, C, T and A Sublanguages . . . 146
3.4 Communication between Rule Components . . . 150
3.4.1 Logical Variables . . . 150
3.4.2 Binding and Using Variables . . . 152
3.4.3 Communication of Results and Variable Bindings . . . 154
3.5 Semantics of Rule Execution . . . 157
3.5.1 Firing ECA Rules: the Event Component . . . 157
3.5.2 The Query Component . . . 160
3.5.3 The Test Component . . . 163
3.5.4 Summary of Event, Query and Test Semantics . . . 163
3.5.5 The Action Component . . . 164
3.5.6 Transactions . . . 165
3.6 Languages for the Event, Query, Test, and Action Components . . . 166
3.6.1 Event Component . . . 166
3.6.1.1 Handling Atomic Events . . . 167
3.6.1.2 Composite Events: Event Algebras . . . 169
3.6.2 Queries in ECA Rules for the Web . . . 174
3.6.3 Tests in ECA Rules for the Web . . . 175
3.6.4 The Action Component . . . 175
3.7 Architecture: Languages and Processors as Resources . . . 175
3.7.1 Rules and Rule Components as Resources . . . 176
3.7.2 Languages as Resources . . . 176
3.7.3 Architecture and Processing: Cooperation between Resources . . . 178
3.8 Normal Form, Evaluation and Optimization [Preliminary] . . . 180
3.9 Implementation Perspectives . . . 183
3.9.1 ECA engine prototype . . . 183
3.9.2 Event Detection . . . 183
3.9.3 Queries and Updates . . . 184
3.9.4 Actions . . . 184
3.9.5 Requirements and Evaluation . . . 185
3.10 Relationship with Existing Languages . . . 185
3.10.1 Triggers on XML Data . . . 185
3.10.2 Triggers on RDF Data . . . 187
3.10.3 ECA Rules on XML . . . 187
3.10.4 ECA Rules in XML . . . 187
3.10.5 Coverage of our Framework . . . 187

A ruleCore and the ruleCore Markup Language (rCML) 197
A.1 A brief overview of ruleCore . . . 197
A.1.1 Architecture . . . 197
A.1.2 Situation detector . . . 199
A.2 The ruleCore Markup Language (rCML) . . . 199
A.2.1 Specification of Event Types . . . 199
A.2.1.1 Basic Events . . . 199
A.2.1.2 Composite Events . . . 200
A.2.2 Specification of Conditions . . . 206
A.2.3 Specification of Actions . . . 207
A.2.4 Specification of ECA Rules . . . 208
A.3 Modelling Use Cases in rCML . . . 209
A.3.1 Use Case 4.2.1 - Changing Phone Number . . . 209
A.3.2 Use Case 4.2.7 - Progress Reports . . . 210
A.3.3 Use Case 4.2.13 - Progress Report Late . . . 213
A.3.4 Use Case 4.2.14 - Polls: Basic Rules . . . 216
A.3.5 Use case 4.2.15 - complex events . . . 218
A.4 Conclusions . . . 219

B Prova language description 221
B.1 Introduction . . . 221
B.1.1 Examples solved with Prova . . . 222
B.1.2 Prova as distributed Agent programming . . . 224
B.1.2.1 Agents Architecture Prova-AA . . . 224
B.1.2.2 Main features of Prova-AA . . . 225
B.1.3 Communicator Prova-AA agent for Java applications . . . 233
B.1.3.1 Specifying the behaviour of Prova-AA agents as state machines 234
B.2 Organising Travels Scenario . . . 236
B.2.1 Initial Planning . . . 236
B.2.1.1 Gathering and reasoning with information . . . 236
B.2.1.2 Arranging the trip according to plan . . . 238
B.2.2 Arranging the trip . . . 239
B.2.3 Adapt Plan to changes . . . 239

Chapter 1

Introduction

The Web and the Semantic Web, as we see it, can be understood as a “living organism” combining autonomously evolving data sources, each of them possibly reacting to events it perceives. The dynamic character of such a Web requires declarative languages and mechanisms for specifying the evolution of the data. This vision of the Web, as well as a state of the art overview of related areas and illustrative use cases, are described in our previous work [2, 3, 51].

Rather than a Web of data sources, we envisage a Web of Information Systems, where each such system, besides being capable of gathering information (querying, both on persistent data, as well as on volatile data such as occurring events), can possibly update persistent data, communicate the changes, request changes of persistent data in other systems, and be able to react to requests from and changes on other systems. As a practical example, consider a set of data (re)sources in the Web of travel agencies, airline companies, train companies, etc. It should be possible to query the resources about timetables, availability of tickets, etc. But in such an evolving Web, it should also be possible for a train company to report on late trains, and travel agencies (and also individual clients) be able to detect such an event and react upon it, by rescheduling travel plans, notifying clients that in turn could have to cancel hotel reservations and book other hotels, or try alternatives to the late trains, etc. Other examples, relevant in this view can be found in our previous deliverable [3].

The importance of being able to update data in the Web has long been acknowledged, and several languages exist (e.g. XUpdate [77], XML-RL [45], XPathLog [48]) for just that. More recently some reactive languages have been proposed, that not only allow for updating Web data as the above ones, but are also capable of dealing-with/reacting-to some forms of events, evaluate conditions, and upon that act by updating data. These are e.g. the XML active rules of [13], of Active XQuery [12], of the Event-Condition-Action (ECA) language for XML defined in [5], and the ECA reactive language RDFTL [58] for RDF data. The common aspect of all of these languages is the use of ECA (declarative) rules for specifying reactivity and evolution. Such kind of rules (also known as triggers, active rules, or reactive rules), that have been widely used in other fields (e.g. active databases [59, 73]) have the general form:

on *event* **if** *condition* **do** *action*

They are intuitively easy to understand, and provide a well-understood formal semantics: when an event (atomic or composite) occurs, evaluate a condition, and if the condition (depending on the event, and possibly requiring further data) is satisfied then execute an action (or a sequence

of actions, a program, a transaction, or even start a process).

In fact, we agree with the arguments exposed for the definition of the above languages in what regards adopting ECA rules for dealing with evolution and reactivity in the Web (declarativity, modularity, maintainability, etc). But in our opinion, these languages fall short in various aspects, when aiming at the general view of an evolving Web as described above. Their events and actions are restricted to updates on the underlying data level; they do not provide for more complex events and actions. In general, actions are more than just simple updates to Web data (be it XML or RDF data). As said above, besides that, actions can be notifications to other resources, update requests of other resources, can result from the composition of simpler actions (like: do this, and then do that), or even transactions whose ACID properties ensure that either all actions in a transaction are performed, or nothing is done. In our view, a general language should cater for such richer actions.

Moreover, events may in general be more than simple atomic events in Web data, as in the above languages. First, there are atomic events other than physical changes in Web data: events may be received messages, or even "happenings" in the global Web, which may require complex event detection mechanisms (e.g (once) any train to Munich is delayed ...). Moreover, as in active databases [26, 79], there may be more composite events. For example, we may want a rule to be triggered when there is a flight cancellation and then the notification of a new reservation whose price is much higher than the previous (e.g. to complain to the airline company). In this respect, there is some preliminary work on composite events in the Web [9], but that only considers composition of events of modification of XML-data in a single document.

We start this report by proposing the language XChange, a language developed in Rewerse for evolution and reactivity on the Web along the lines just exposed. XChange is a high level language for programming reactive behaviour and distributed applications on the Web. It provides advanced features such as propagation of changes for evolution on the Web, event-based communication between Web-sites, composite events, complex updates over Web data, and complex actions and transactions. Moreover, the language is completely in line with the query language Xcerpt, that is being developed in Rewerse within WP I4.

The presentation of XChange in this report comprises the description of the language, including details of each of the events, conditions and action parts, its declarative and operational semantics. Moreover, the language is illustrated in a number of use cases chosen from our previous deliverable I5-D2 [3].

After the presentation of XChange, we proceed by proposing a general framework and architecture for dealing with the challenges posed by evolution and reactivity in the Semantic Web, namely in what concerns the abstraction levels of behaviour and the heterogeneity of languages and behaviour (besides of data).

A main goal of the *Semantic Web* is to deal with the heterogeneity of data formats and languages and provide unified view(s) on the Web, as an extension to today's *portals*. In this scenario, XML (as a format for storing and exchanging data), RDF (as an abstract data model for states), OWL (as an additional framework for state theories), and XML-based communication (Web Services, SOAP, WSDL) provide the natural underlying concepts. The Semantic Web does not possess any central structure, neither topologically nor thematically, but is based on peer-to-peer communication between autonomous, and autonomously developing and evolving nodes. This *evolution* and *behavior* depends on the cooperation of nodes. In the same way as the main driving force for the Semantic Web idea was the heterogeneity of the underlying data, the heterogeneity of concepts for expressing behaviour requires an appropriate handling on the semantic level. When considering evolution, the concepts and languages for describing

and implementing behavior will surely be diverse, albeit due to different needs, and it is unlikely that there will be a unique language for this throughout the Web. Since the contributing nodes are prospectively based on different concepts such as data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that the *concepts* and the actual *languages* are independent. As such, besides having a single concrete language for dealing with evolution and reactivity as the presented XChange, the Semantic Web calls for the existence of a general framework able to deal with this heterogeneity of languages.

Our view is that this general framework should be based on a general ECA language that allows for the usage of different event languages, condition languages, and action languages, considering ontological descriptions and mappings for these languages. Each of these different (sub)languages should adhere to some minimal requirements (e.g. dealing with variables), but it should be as free as possible.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general, especially if one wants to reason about evolution, ECA rules (and their components) must be communicated between different nodes, and may themselves be subject to being updated. For that, the ECA rules themselves must be represented as data in the Semantic Web. This need calls for a (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [10], but it does not tackle the internal structure of events, actions, and the generality of rules, as described here. Moreover, to deal with the requirements of heterogeneity and of reasoning about rules, an ontology of ECA rules and (sub)ontologies for events, conditions and actions, with rules possibly specified in RDF/OWL, is required. Additionally, the rule components have to be related to actual languages and processors.

In this report, we propose a general model and language for evolution and reactivity in the Semantics Web, which caters for the just exposed requirements of heterogeneity, and sketch a markup for active rules, that will be the basis for a (near) future discussion with the WPI1 of Rewerse (Rule Markup) in order to establish the final markup proposal. Importantly, this language fully complies with XChange, and this is made clear along with the proposal, in that XChange events, conditions and actions can be used as sublanguages of the framework. It is also shown how other languages for events, condition and actions can be incorporated in this general view. Namely, we illustrate it with a language for events using an event algebra similar to that of SNOOP [25], with a simple language for actions, and with the ruleCore framework [62]. Although not developed within Rewerse, the ruleCore system resulted from joint work between industry and an academic partner (University of Skövde) of Rewerse, and the study of its possible relation to the work exposed in this report was done within Rewerse. Appendix A gives a brief description of ruleCore, and the corresponding markup language rCML. This appendix also includes, for illustration purposes, and for helping contrasting the languages, the encoding in rCML of the selection of use-cases [3] considered in the chapter on XChange.

The description of the general framework is accompanied in this report by the sketching of a global architecture and considerations for the implementation of the framework to be done in the next future phase of the work in this WP. In this future phase, when building prototypes, it will be especially important to find adequate languages that serve as vehicle for the implementation. In this respect, the language Prova [43], also developed within Rewerse mainly by partner Dresden, seems to have the adequate ingredients for that purpose, as it will be made clear. In another appendix we briefly describe Prova with special emphasis on the features that make it appropriate for implementing evolution and reactivity in the Semantic

Web. In this description, for illustration we also include an implementation of a use case from [3].

The report ends, apart from the mentioned appendices, with a brief comparison with the existing languages for active rules for XML and RDF, including the XChange language and ruleCore, and show how they are covered in the general framework, and where they can be useful in the architecture.

Chapter 2

The Reactive Web Language XChange

Today's World Wide Web (WWW, or Web for short), undoubtedly by far the largest information system, is largely passive and provides only limited support for reactivity: mostly it only is a collection of data — stored primarily in HTML or XML documents — that can be retrieved and viewed upon request. With the emergence of new Web applications in electronic commerce, business-to-business, logistics, and information systems for biological data, reactivity on the Web receives increasing attention.

XChange aims at, at least partly, filling the gap between the current, passive Web and the need for reactivity. XChange is a high-level language for programming reactive behavior and distributed applications on the Web. It provides advanced, Web-specific features such as propagation of changes (evolution) on the Web (*change*) and event-based communication (reactivity) between Web-sites (*exchange*), hence also its name, XChange.

In this chapter we present the language, and the declarative and operational semantics of XChange. The presentation starts by discussing the paradigms XChange is built upon (Section 2.1). As argued in the introduction, one of these paradigms is the use of Event-Condition-Action rules (for short, ECA-rules), hence we continue by presenting each of the three rule components consecutively: Events and Event Queries, which detect situations of interest that require a reaction (Section 2.2); Conditions, which are Web Queries based on the query language developed in the Rewerse working group I4 (Section 2.3); Actions, which can update Web resources or raise new, application-dependent, events (Section 2.4). Each of the components can be used separately from the other components, e.g., in the general framework described in the next chapter; however used together as the components of XChange ECA-rules, they form a homogeneous rule-based reactive language with a clear design based on pattern queries, which is described in Section 2.5. Finally, we illustrate XChange rules with a set of selected use cases from [3] (Section 2.6).

More detailed information on XChange can be found at [61], while [4] provides a very brief and abstract overview of XChange's aims and paradigms.

2.1 Paradigms of XChange

Clear paradigms that a programming language follows provide a better language understanding and ease programming. Moreover, explicitly stated paradigms are essential for Web languages, since these languages should be easy to understand and use also by novice practitioners. This section introduces the paradigms upon which the language XChange relies.

2.1.1 Event vs. Event Query

An *event* is a happening to which each Web site may decide to react in a particular way or not to react to at all. Typical events are the insertion, deletion, or modification of a data item in some data source, e.g., an element in an XML document. Events can also be higher-level application-dependent happenings, e.g., in a tourism application we can find events like "cancellation of flight UA917 for passenger John Q Public." In order to notify Web sites about events and to process event data, events need to have a data representation.

Event queries are queries against event data. Event query specifications differ considerably from event representations (e.g. event queries may contain variables for selecting parts of the events' representation). Most proposals dealing with reactivity do not differentiate between event and event query. Overloading the notion of event precludes a clear language semantics and thus makes the implementation of the language and its usage much more difficult. Event queries in XChange serve a double purpose: detecting events of interest and temporal combinations of them, and selecting data items from events' representation. Variables are used in event queries as place holders for data items that are to be used in the other parts of XChange rules.

2.1.2 Volatile vs. Persistent Data

The development of the XChange language – its design and its implementation – reflects the view over the Web data that differentiates between *volatile data* (event data communicated on the Web between XChange programs) and *persistent data* (data of Web resources, such as XML or HTML documents). Web data is in general *persistent*, *modifiable* (can be updated), and has to be retrieved in a *pull*-manner. In contrast, event data is *volatile*, *unmodifiable* (cannot be updated), and is usually received in a *push*-manner.

The distinction of persistent (Web) data and volatile (event) data can be illustrated with a metaphor. Persistent data is like (computer-)*written text*. Once produced, it is available permanently for anyone to read (or rather anyone who is allowed to do so). Later, the text can be modified directly by editing it. Volatile data is like *spoken words*. Once a sentence is spoken, its information is available only to the listeners and only as long as they remember. A spoken sentence cannot be changed: the only way to correct, complete, or invalidate its information then is through speaking new sentences.

XChange's language design enforces a clear separation of volatile and persistent data and entails new characteristics of event processing on the Web (discussed later in Section 2.2.5). However, in applications where a part of the volatile data received by Web sites needs to be stored for a long time or forever, the data of interest can be easily made persistent. Moreover, the language XChange is flexible enough (in terms of language design and program evaluation) for adapting it to other kinds of application domains as the ones intended to be primarily solved by this proposal.

2.1.3 Rule-Based Language

Reactivity can be specified and realised by means of reactive rules [28, 60, 74]. XChange is a rule-based language that uses (i) reactive rules for specifying the desired reactive behaviour (cf. Section 2.5) and (ii) deductive rules for constructing views over Web resources' data.

An XChange program is located at one Web site and contains one or more reactive rules of the form *Event query – Web query – Action*. Every incoming event is queried using the *event query* (query against volatile data). If an answer is found and the *Web query* (query to persistent data) yields also an answer, then the *Action* is executed. The fact that the event query and the Web query have answers (i.e. evaluate successfully) determines the rule to be fired; the answers influence the action to be executed, as information contained in the answers are generally used in the action part.

XChange embeds the Web query language Xcerpt, which is being developed in parallel by the REWERSE working group I4 [66], for expressing the *Web query* part of reactive rules and for specifying deductive rules in XChange programs. Note that Xcerpt is deployed also in querying single occurrences of incoming events. Xcerpt (deductive) rules allow for constructing views over (possibly heterogeneous) Web resources that can be further queried in the *Web query* part of XChange reactive rules. Not only integration and restructuring of persistent data is possible with Xcerpt, but also reasoning with persistent data (given e.g. in XML or RDF format).

Complex reactive applications can be elegantly implemented in XChange, as rules are means for structuring complex programs. To illustrate just this, in Section 2.6 we show how to implement a selection of use cases from [3].

2.1.4 Pattern-Based Approach

XChange is a *pattern-based language*: event queries, Web queries, event raising specifications, and updates describe *patterns* for events requiring a reaction, Web data, raising event messages, and updating Web data, respectively. Patterns are templates that closely resemble the structure of the data to be queried, constructed, or modified.

Not mixing patterns for data to be constructed, e.g. for insertion in a given document, with paths for selecting data items or for specifying e.g. where new data is to be inserted, the programmer needs to understand and use one single concept — that of data pattern. This uniform specification allows for an easier programming, also because the overall structure of XChange programs is easy to grasp.

2.1.5 Transactional Reactivity

Complex Updates. XChange supports the specification and execution of *simple updates*, i.e. insertions, deletions, and replacements of persistent data items, such as XML or RDF data. *Complex updates* expressing ordered or unordered conjunctions, or disjunctions of (simple or complex) updates are also offered by XChange. Such updates are required by real applications. E.g. when booking a trip on the Web one might wish to book an early flight *and* of course the corresponding hotel reservation, *or* else a late flight *and* a shorter hotel reservation. The application scenarios given in the Use Case deliverable of this working group [3] have motivated the need for executing such complex updates in an *all-or-nothing manner*. For this, XChange supports a concept of transactions [71].

Transactions and ACID Properties. XChange transactions obey the ACID properties, i.e. Atomicity, Consistency, Isolation, and Durability. Atomicity and isolation are considered in XChange, the issues of consistency and durability for transactions are currently not investigated in the project. XChange will build on standard solutions from database systems that need to be adapted to the Web.

Transactional Events. Transactional events (i.e. commit, abort, request) are offered by XChange. They are needed for supporting transactions.

2.1.6 Communication Paradigms

Peer-to-Peer. In XChange, the *peer-to-peer* communication model is used for communicating event data between Web sites. This means that all parties have the same capabilities and every party can initiate a communication session. Event data is directly communicated between Web sites without a centralised processing of events. XChange assumes no instance controlling (e.g. synchronising) communication on the Web.

Push Strategy. For communicating (propagating) events on the Web, two strategies are possible: the *push* strategy, where a Web site informs possibly interested Web sites about events, and the *pull* strategy, where interested Web sites query periodically (poll) persistent data found at other Web sites in order to determine changes. For propagating events (i.e. communicating data about events), a push strategy has several advantages over a strategy of periodically polling: It allows faster reaction to events, as a notification is communicated as soon as possible as opposed to a detection at the next periodical pull. It saves resources, both locally and on the network. Locally, a client interested in some change of Web data does not have to store the old Web page to detect differences (changes) from the new version. On the network, a push strategy can reduce network traffic, since communication only takes place when a change has happened, and only the changes in information have to be communicated.

The pull strategy is already supported by languages for Web queries like XQuery or Xcerpt that query persistent data. Therefore, so as to complement the framework, XChange offers the *push* strategy. The push strategy requires event queries to be incrementally evaluated by so-called *event managers*. In the case of XChange, this is done at every XChange-aware Web site.

Communication Protocol. The language XChange is not dedicated to a particular communication protocol, instead its high-level nature allows for implementing distributed, reactive applications following different rules for communicating data. However, its goal is to realise reactivity on the Web or Semantic Web and thus, event data is communicated over the HTTP protocol.

2.1.7 Composite Events Defined through Event Queries

Composite event queries allow to recognise temporal patterns over incoming events – to recognise *composite events*. In general, composite events are obtainable with an event algebra, comprising various event operators, as will be further detailed in Chapter 3. XChange's (occurrences of) *composite events* are defined through *composite event queries* (see Section 2.2.3.3) – they are *answers* to composite event queries, which include event operators similar to those of

event algebras. E.g. an XChange event query can ask for occurrences of an increase of share values by more than 5 percent for the company Siemens, *followed* by an increase of share values for the company SAP on the stock market. An answer to such an event query contains instances of the two specified component event queries (i.e. increase of share values). Another XChange event query can ask for *all* stock market reports that have been registered between the occurrences of an increase of share values for the two mentioned companies. An answer to such an event query contains, besides the instances of the events signaling an increase for the shares of the companies, all reports registered between these two instances.

2.1.8 Processing of Events

Local Processing. No central processing of event queries is assumed as such an approach is not suitable on the Web. Instead, event queries are processed locally at each Web site[1]. Each such Web site has its own local *event manager* for processing incoming events and evaluating event queries against the incoming event stream (volatile data), and for releasing event query instances after a finite time.

Incremental Evaluation. Event queries need to be evaluated in an *incremental* manner, as data (events) that are queried are received in a stream-like manner and are not persistent. For every incoming event that might be relevant to a reactive Web site and could contribute as a component to an event query instance specified in the rules of the Web site's reactive program(s), a partial *instantiation* of the involved event queries is realised. An instance of a specified composite event query is detected when instances for all specified component event queries have been detected.

Bounded Event Lifespan. An essential aspect for event processing is that each reactive Web site controls its own event memory usage. In particular, which events and for how long they are kept in memory depends only on the event queries posed at a Web site. Neither Web queries nor event queries posed at other Web sites can influence the event lifespan (i.e. the time period an event is kept in memory). Event lifespans are automatically determined from the event queries already registered at a Web site.

Event queries need to be in such a way that no data on any event can be kept forever in memory, i.e. the event lifespan should be *bounded*. Keeping all events in memory is not a suitable approach for reactivity on the Web; the amount of events a Web site receives might be huge, causing a continual growth in storage requirements. By language design, XChange event queries are such that volatile data remains volatile. This is consistent with the clear distinction between events as volatile data and Web resources' data as persistent data. However, making *part* of the incoming event stream persistent is application-dependent (for example applications where statistics over data of incoming streams play a role).

2.1.9 Relationship Between Reactive and Query Languages

A working hypothesis of XChange is that a reactive language for the Web should build upon, or more precisely embed, a Web query language. There are two reasons for this. First, specifications of reactive behaviour often refer to Web contents, which requires querying. Second,

[1]Below, in Section 3.7, in the context of the general framework, we consider the possibility of delegating the processing of event detection to another site providing such a service. Anyway the processing is still local.

reactive behaviour necessarily refers to (more or less recent) events, here expressed as event queries. For these two tasks, XChange follows a uniform approach where the languages used for querying Web contents and for querying events are as close as possible to each other. Note, however, that querying events calls for additional (mainly, temporal) constructs that are not needed for querying Web contents. For example, the interest in a conjunction of events that occur in a given time interval can not be specified by means of a Web query language; on the other hand, for querying data of Web resources temporal relations between parts of the data are not needed.

The query language embedded in XChange is the Web query language Xcerpt [65, 64, 66]. In the framework of XChange, Xcerpt's capabilities are deployed for querying and reasoning about Web resources' data, and for querying (single) occurrences of events. A prominent reason for choosing Xcerpt over other existing query languages for embedding it into XChange includes the pattern-based approach followed in querying and constructing Web data, and the fact that simulation unification can be deployed not only for querying persistent data but also for querying single occurrences of events. Moreover, Xcerpt uses the notion of logical variables, and these provide a very clear means of communication between the different parts of an XChange ECA rule. However, as we shall see in the general framework, for a combination with other query languages the usage of logical variables for linking the various parts is also important (cf. Section 3.4).

2.1.10 Language Syntax

The development of XChange followed the conviction that a language for the Web should have three syntaxes: a *compact human-readable* syntax, a *machine-processable* (XML) syntax, and a *visual* syntax. The compact human-readable syntax should be as compendious as possible and easy to use by programmers. The XML syntax is desirable for interchanging programs and manipulate them with XML-based tools and languages (e.g. to query and modify them). A visual language "can greatly increase the accessibility of the language, in particular for non-experts"[17]. However, programmers have the freedom to choose whichever syntax they prefer.

At present, the language XChange has a compact human-readable syntax (which is a term-based syntax where a term represents a Web document, a query pattern, an event pattern, or an update pattern) and an XML syntax. The development of a visual counterpart of XChange's textual language is sought for. Along this line, the visual rendering of Xcerpt programs – visXcerpt [6] – is to be extended.

Notes. Along this chapter, the introduction of language constructs will be accompanied by the corresponding syntax rules. They explicitly state the valid combinations of XChange language constructs. Thus, the grammar for the language XChange is constructed in a stepwise manner. One of the most commonly used meta-syntactic notations for specifying the syntax of programming languages is the Backus-Naur Form (BNF) [54]. The BNF notation is a formal metasyntax used to express context-free grammars. There are many extensions to the BNF notation, one of them is the Extended Backus-Naur Form (EBNF) [75] notation. A short explanation of the EBNF notation follows (for a more detailed introduction into the EBNF notation, see [75]): Terminals are symbols or words in the language, nonterminals are units representing a grammatically correct sequence of terminals. Productions are defined that specify the valid ways how nonterminals can be replaced by terminals and other nonterminals. Extensions to BNF include: `|` denotes disjunction, `*` denotes that the preceding symbol or parenthesised ex-

pression may occur zero or more times, + denotes that the preceding symbol or parenthesised expression may occur one or more times, [ ] denotes optionality (note that instead of square brackets, the symbol ? is used so as not to confuse the optionality with the Xcerpt syntax for total specifications). (These extensions can be expressed in BNF by using extra productions.) *Note* that keywords are shown as quoted strings of characters (like "`and`").

2.2 XChange Events and Event Queries

2.2.1 Events

The notion of *event* is defined as "something that happens at a given place and time"[2]; this explanation adapted from the WordNet 2.0[3] lexical database for the English language has (slight) variations that are used in different domains. In physics, an event is a change in the state of the world; in relativity theory, the fundamental observational entity is the event, i.e. a phenomenon located at a single point in space-time. In event-driven programming, "an event is a software message that indicates something has happened"[4]. For example, graphical user interface programming follows this paradigm, where small programs called event handlers are to be called in response to external events. Considering the public relations domain, an event is a means for establishing and promoting a favourable relationship with the public; it can be organised as workshops, exhibitions, or panels that, in general, have a particular topic.

Section 2.1.1 already introduced the notion of *event* in an informal manner; it also stressed the fact that one can conceive any kind of events in XChange. The above discussion on events does not offer a precise definition either, as might have been expected. It just states that a large number of events are conceivable and they correspond to many application domains. Thus, the (very general and abstract) definition of event fits perfectly into a reactive language for the Web. However, for realising reactivity on the Web, events require some representation for communicating their data between reactive programs on the Web and for processing their data by (local) event managers. In XChange, events are represented as XML documents. The language XChange has the ability to send, receive, and query events that are represented as *XChange event messages* (discussed in Section 2.2.2), i.e. messages containing any kind of event data represented as XML.

A *situation* is a combination of circumstances, that is a combination of events and other situations. Situations reflect particular states of the world, from low-level (such as ordered conjunctions of update operations on XML documents) to high-level ones (such as flight cancellations for which the airline does not grant an accommodation). The preceding Use Case Deliverable [3] has motivated the need for detecting situations that occur on the Web and recognised the ability to detect them as a requirement for Web reactive languages.

Not all events that have occurred on the Web, not all possible combinations of them (forming situations) are *of interest* for a Web site. Of interest are events and situations whose detection requires an action to be automatically executed. At a moment in time, each Web site is interested in some classes of events and situations; these classes are determined by the rules of the Web site's reactive program (cf. Section 2.1.3). Modifying, deleting, or specifying new rules in a reactive program might entail other classes of events or situations to be of interest.

[2]Word Reference, `http://www.wordreference.com/definition/event`
[3]WordNet 2.0, `http://wordnet.princeton.edu/`
[4]Labor Law Encyclopedia, `http://encyclopedia.laborlawtalk.com/Event`

Table 2.1: XChange Atomic Events

<table>
<tr><td rowspan="5">atomic events</td><td colspan="2">explicit events
(event messages)</td></tr>
<tr><td rowspan="4">implicit</td><td>updates</td></tr>
<tr><td>queries</td></tr>
<tr><td>transactional events</td></tr>
<tr><td>system events</td></tr>
</table>

This section continues with presenting the two kinds of events supported by XChange, namely *XChange atomic events*, which reflect *events* as discussed just previously (Section 2.2.1.1) and *XChange composite events*, which reflect *situations* (Section 2.2.1.2). Subsequently, Section 2.2.1.3 discusses the occurrence time for atomic events and composite events, respectively.

2.2.1.1 Atomic Events

An XChange *atomic event* is an event as it has been introduced in the previous section. XChange distinguishes between two kinds of atomic events: *explicit* events and *implicit* events. *Explicit events* are explicitly raised by a user or by a (predefined) XChange program at a Web site and sent to other Web sites through *event messages* (see Section 2.2.2). *Implicit events* are events that occur locally at a Web site (e.g. local updates of data or system clock events). Implicit events have also a representation as event messages, but a more simplified one than for explicit events. Implicit (or explicit) events raised and sent from a Web site to another become explicit.

The kinds of *atomic events* considered in XChange are presented in Table 1. An *update* executed or a *query* posed locally at a Web site are for XChange local events, i.e. raised at this Web site and processed at this Web site. *Transactional events* (e.g., commit, abort) are local events needed as XChange supports the concept of transactions (cf. Section 2.1.5). *System events* (e.g. system clock events) are events that are coming from the encompassing "system" and might be useful to handle together with explicit and/or implicit events. A system event might be explicit or implicit, depending whether or not it is transmitted from one Web site to another.

Remote events, i.e. events informing a Web site of queries, updates, transactional or system events or of any other (application specific) matter, are always explicit and are expressed through event messages.

For further discussion on atomic events, in particular the need for application and semantic level events for dealing with the Semantic Web, see Section 3.1.2.

2.2.1.2 Composite Events

XChange composite events reflect situations (introduced in Section 2.2.1). Composite events express temporal relationships between atomic events that have occurred on the Web. Also, they can express the non-occurrence of some events while other events have occurred. XChange's (occurrences of) *composite events* are defined as answers to *composite event queries* (see Section 2.2.3.3). For understanding XChange's composite events and their representation, an elaborate discussion on XChange's event queries is needed. Section 2.2.3 elaborates on event queries in

XChange. Thus, a more detailed discussion on XChange's composite events is postponed to Section 2.2.3.5.

2.2.1.3 Events' Occurrence Time

The occurrence time of (atomic or composite) events plays an essential role when determining whether to react or not to incoming events. For example, one might want to react to a certain class of atomic events but only if they are received before a given time point. For composite events, the occurrence time of atomic events is used for determining if a certain temporal order between them is met or not. Moreover, based on the occurrence time of events, atomic and partially detected composite events can be released after a bounded time (see Section 2.2.3.3).

Atomic Events. An XChange atomic event occurs at a *point in time*. The occurrence time of an atomic event is the time point at which its representation has been *received* by a Web site. The occurrence time of an explicit event can not be considered as the time point at which its representation has been sent, since the Web lacks a global time and the processing of events is done locally at each Web site. *Note* that the same atomic event sent to different Web sites may have different occurrence time at its recipients.

Composite Events. In general, more than one atomic event are used to answer an XChange composite event query. Thus, XChange composite events have (in general) more than one constituent (atomic) events (each of them having its own occurrence time).

The work done in the active database field considers that each composite event has an occurrence time point, like atomic events do. The occurrence time of a composite event is the occurrence time point of the last received constituent atomic event. Queries against incoming events specifying e.g. that a particular event should occur two times during the occurrence of a composite event do not express the programmer's intuition. Thus, XChange follows another approach: XChange composite events do not have an occurrence time point, instead they stretch over time (they have a duration). Each XChange composite event has a *beginning time* and an *ending time*. In general, a composite event inherits from its components a beginning time (i.e. the reception time of the first received constituent event that is part of the composite event) and an ending time (i.e. the reception time of the last received constituent event that is part of the composite event). This is not the case for all composite events. Recall that composite events are defined as answers to composite event queries. Consider now a composite event query asking for non-occurrences of instances of event query EQ during a given time interval. Answers to such a composite event query do not have any constituent atomic events (as it is asked for non-occurrence); they have just a beginning and an ending time, time points determined by the given time interval. For each kind of XChange composite event queries, the beginning and ending time of their instances are explicitly specified in Section 2.2.3.3.

2.2.2 Event Messages

Web sites are interested not only in events that have occurred locally but also in events that have occurred at other Web sites (remote events). Information about events that have occurred on the Web needs to be communicated to possibly interested Web sites.

Event messages communicate event data between (same or different) Web sites. More concretely, XChange programs found at different Web sites raise events and send their repre-

sentations (i.e. event messages) to one or more XChange programs. For gaining the flexibility needed for implementing different kinds of applications and for coping with event data having a complex and irregular structure, the XML format has been chosen for representing event messages.

Communication of event messages follows a push strategy (cf. Section 2.1.6), i.e. Web sites *inform* other Web sites about (implicit or explicit) events that have occurred on the Web. XChange excludes broadcasting of event messages on the Web (i.e. sending event messages to all sites of a portion of the Web), since indiscriminate sending of event messages to all Web sites introduces problems for a non-centrally managed structure such as the Web. Thus, in XChange each event message has a determined recipient Web site.

A question arises: *How does a Web site know which Web sites are interested in which kind of events?* In this chapter we assume that a (kind of) *subscription mechanism* exists, a procedure through which Web sites are made aware of correspondences between Web sites and classes of events to be notified of. Note that Web sites do not always need to explicitly subscribe to (classes of) events of interest. Instead, subscription knowledge might be implicit. For example, reservations made for a particular flight contain implicitly the interest in notifying the passengers (perhaps by sending notifications to their personalised organisers) about delays or cancellations of the respective flight. Subscribing to (classes of) events is not a complex mechanism; different applications might use different subscription mechanisms. Thus, the rest of this report abstracts away from a particular subscription mechanism by assuming that Web sites do have the necessary subscription information.

An XChange event message contains information about its sender Web site. This piece of information might be important for the recipient Web sites. Assume that Mrs. Smith is on vacation. Though, she would like her personalised organiser to be notified by her secretary if important problems occur in one of Mrs. Smith's projects. In such cases, not only the content of the event message but also the sender plays an important role in detecting the desired situation. However, applications not always need to make use of the sender address. Thus, the event language component of a reactive language should offer the ability to express partiality in queries to event data (i.e. to leave out parts of the event messages that are not of interest when specifying patterns to them). Section 2.2.3 shows that this language requirement is fulfilled by XChange.

XChange event messages have two time stamps: one denoting the time point at which the sender has raised the event whose representation the message is, and one denoting the time point at which the recipient has received the event message. Time stamps play an essential role in determining temporal combinations of events and in filtering out event messages that have not been received in a time interval of interest. Thus, (reception) time stamps allow for detecting complex situations of interest.

Discussion. The time stamp an event message gets from the sender might be the raising time or the sending time of the event message. XChange event messages use the raising time, i.e. the time at which the construction of the event message has been finalised, the event message being now ready for sending. The sending time of an event message might be also useful to applications, case in which a slight modification of the XChange prototype should be used.

The sender and recipient Web sites' addresses, the two time stamps of event messages are *parameters* included in the representation of XChange event messages. The next section discusses the parameters of event messages in more detail.

2.2.2.1 Parameters of Event Messages

An XChange *event message* is an XML document with a root element labelled `event` and at least five child elements labelled `raising-time`, `reception-time`, `sender`, `recipient`, and `id`. The design decision of representing the parameters of event messages as child elements and not as attributes of the event message's root is that they may contain complex content. For example, the time point of raising an event message at a Web site might be represented as a time point accompanied by a specification of the calendar used in the respective country. Thus, an event message wraps the event data like in the following:

```
<?xml version="1.0"?>
<xchange:event xmlns:xchange="http://xcerpt.org/xchange">
  <xchange:sender> sender </xchange:sender>
  <xchange:recipient> recipient </xchange:recipient>
  <xchange:raising-time> raising-time </xchange:raising-time>
  <xchange:reception-time> reception-time </xchange:reception-time>
  <xchange:id> id </xchange:id>
  event data
</xchange:event>
```

where

- `sender` is the URI of the Web site where the event has been raised, that is its representation as event message has been constructed and sent to one or more Web sites. The URI of the sender is determined and inserted into the event message by the Web site's event manager before sending it.

- `recipient` is the URI of the Web site that received the event message. As already explained, XChange excludes broadcasting of event data, implying that the recipient Web site(s) of an event message must be known before sending it. For an event message, at least one recipient needs to be given in the event message specification used to raise the event.

- `raising-time` is the time of the event manager of the Web site raising the event (local time of that machine).

- `reception-time` is the time at which the event manager of the recipient Web site receives the event message (again, the local time of that machine). *Note* that the reception time of an event message might be "before" its raising time as no global time exists on the Web (proposals exist, such as [67], for models of an approximated global time base for distributed systems; however this is not realistic in the largest distributed system – the Web).

  Event messages' time stamps (raising time and reception time) are given in XChange by using the ISO 8601 standard format for the representation of dates and times[5].

- `id` is an event message identifier given at the recipient Web site. Each event message gets at its reception such an identifier for uniquely identifying it in querying. The format and the method (e.g. counting received event messages each day and identifying them

[5] ISO 8601, `http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html`

with the temporal specification of the current day followed by the counter) used for event messages' identifiers are application-specific. The current implementation of the XChange event manager uses positive integers for identifying event messages.

- `event data` is an XML element (having possibly complex structure) containing information about an event that has occurred. How the data about the event is represented is application-specific. Reactive applications communicating through XChange event messages are not restricted to XML-based applications. For example, XML-serialised RDF data can be also communicated between and processed by XChange programs.

XChange assumes that each event message has a distinctive reception time, i.e., at each point in time only a single event message is received. However, for extensions or future versions of the language where this assumption is lifted, the event `id` parameter uniquely identifying event messages has been introduced. Whether this assumption is lifted or not depends on the time granularity used for event reception.

A DTD for the XML representation of event messages is given next. The namespace prefix chosen for the DTD is the one used throughout this chapter (prefix `xchange` for namespace `http://xcerpt.org/xchange`). A parameter entity is used in place of the content of an event messages; the content (event data) is application specific and thus has to be defined depending on the application.

```
<!DOCTYPE xchange:event [
   <!ELEMENT xchange:event (
              xchange:sender, xchange:recipient,
              xchange:raising-time, xchange:reception-time,
              xchange:id,
              %event-data)>

   <!ATTLIST xchange:event xmlns:xchange CDATA #FIXED
             "http://xcerpt.org/xchange">

   <!ELEMENT xchange:sender              (#PCDATA)>
   <!ELEMENT xchange:recipient           (#PCDATA)>
   <!ELEMENT xchange:raising-time        (#PCDATA)>
   <!ELEMENT xchange:reception-time      (#PCDATA)>
   <!ELEMENT xchange:id                  (#PCDATA)>
]>
```

Being XML documents, XChange event messages can also be represented as Xcerpt data terms [64] and thus, methods developed for querying persistent data can be also applied for querying incoming event messages. The importance of this note will be made clear in Section 2.2.3. The examples given in this section use the term syntax to represent event messages, but the reader should keep in mind that XChange programs communicate through XML documents that represent event messages.

Example 2.1 (XChange Event Message Notifying an Exhibition) *The XChange event message below is sent by* `http://artactif.com` *informing the travel organiser of Mrs. Smith about an exhibition of the painter G. Barthouil. Note the use of the* `xchange` *namespace for the keyword* `event` *and for the parameters of an XChange event message. Note also that the*

examples abstract away from a particular communication protocol. Here, `organiser` *denotes the communication protocol used by a personalised organiser.*

```
xchange:event {
  xchange:sender {"http://artactif.com"},
  xchange:recipient{"organiser://travelorganiser/Smith"},
  xchange:raising-time {"2005-05-05T10:15:00"},
  xchange:reception-time {"2005-05-05T10:17:00"},
  xchange:id {"5517"},
  exhibition {
    painter {"G. Barthouil"}, location {"Marseille"},
    time-interval{"[2005-05-08..2005-05-18]"},
    visit-hours { from {"10:00"}, until {"18:00"}}
   }
 }
```

An event message is an envelope for an arbitrary XML content. Thus, multiple event messages can (but not necessarily) be nested making it possible to create trace histories.

Example 2.2 (Nesting XChange Event Messages) *Mrs. Smith notifies a friend of her about G. Barthouil's exhibition. The following XChange event message is sent by Mrs. Smith's travel organiser and contains the received event message from the previous example:*

```
xchange:event {
  xchange:sender {"organiser://travelorganiser/Smith"},
  xchange:recipient{"organiser://travelorganiser/myFriend"},
  xchange:raising-time {"2005-05-06T11:10:20"},
  xchange:reception-time {"2005-05-06T11:11:20"},
  xchange:id {"5611"},
  content {
    xchange:event {
      xchange:sender {"http://artactif.com"},
      xchange:recipient{"organiser://travelorganiser/Smith"},
      xchange:raising-time {"2005-05-05T10:15:00"},
      xchange:reception-time {"2005-05-05T10:21:20"},
      xchange:id {"1234"},
      exhibition {
        painter {"G. Barthouil"}, location {"Marseille"},
        time-interval{"[2005-05-08..2005-05-18]"},
        visit-hours { from {"10:00"}, until {"18:00"} }
     }
    }
  }
 }
```

Note that XChange messages are compatible with the messages and the "message exchange patterns" of SOAP [76]. XChange event messages can be seen as a very simplified form of

SOAP messages, as only the minimum information is required (time stamps, sender, recipient, and id). However, XChange does not preclude the usage of more complex parameters of event messages. XChange applications can be implemented in such a way to construct such event messages and to understand their meaning properly.

2.2.2.2 Representation of Implicit Events

As explained in Section 2.2.1.1, implicit events are events that occur locally at an (XChange-aware) Web site. They become explicit if their representation is sent as an event message to other Web sites. However, at the level of event representation no differentiated treatment should be applied for the two kinds of events. A uniform way of representing events is a premise for using the same event language for querying them. Thus, XChange uses *event messages* for representing not only explicit events but also implicit events.

An event message representing an implicit event needs to reflect the *type* of change in the state of the world (e.g. timer events, updates, transaction-related events). Clearly, the content of the event message is tailored to the event type it represents. Different approaches are conceivable for event messages to carry this information, for example:

(a) the event type is represented as label of the root (e.g. `xchange:timer-event` for timer events) and the content represents other relevant information (e.g., for reacting on 2005-06-12T11:15, `time{"2005-06-12T11:15"}`),

(b) the event type is represented (e.g. `xchange:type{"timer-event"}`) as an event message parameter and the content representation is like for the case above.

On the other hand, some of the parameters of event messages can be suppressed – the *sender* and the *recipient* are the same Web site, i.e. the Web site where the event occurred and where it is first processed. Thus, there is no single possible representation of implicit events as event messages. One needs to decide which representation is better suited for the intended applications and to modify the XChange runtime system accordingly.

Example 2.3 (XChange Implicit Event Representation) *The following XChange event message gives a representation of an implicit event representing a modification of a Web resource; the event has occurred at Web site `http://xcerpt.org/xchange/`. The sender and recipient of the event message are not contained in the representation; the parameter `xchange:-type` denotes the type of the implicit event, here an (insertion) `update`.*

```
xchange:event {
  xchange:reception-time {"2005-05-05T10:15:00"},
  xchange:type { "update" },
  insertion {
    resource { "http://xcerpt.org/xchange/news.xml" },
    term {
       article{
          title { "Reactivity on the Web" },
          subtitle { "Paradigms and Applications of the Language XChange" },
          proceedings-of { "SAC'2005" }
       }
    },
```

```
    parent { news {} }
  }
}
```

2.2.3 Event Queries

For detecting situations that have occurred on the Web and require a reaction to be automatically executed, incoming event messages (i.e. representations of events that have occurred on the Web) need to be queried. Section 2.1.2 has pointed out differences between (volatile) information about incoming events and (persistent) information of Web resources, recognising that Web query languages are not suitable for querying event data. For this reason, XChange offers *event queries* – queries against event data.

Detection of real life situations often needs not only just one event to occur, but in general takes several events into account. The temporal order of these (component) events has to be taken into account, and there might be further restrictions regarding occurrence time points. Mirroring these practical requirements, XChange offers not only *atomic event queries* but also *composite event queries*. Thus, an XChange event query (symbol `EvQ`) is either an atomic event query (symbol `At_EvQ`) or a composite event query (symbol `Comp_EvQ`):

```
EvQ ::= At_EvQ | Comp_EvQ
```

where atomic and composite event queries are defined in the following sections.

This section on event queries is structured as follows: we start by stating explicitly essential traits of XChange event queries; their introduction intends to ease the understanding of most of the design decisions of the XChange event language. We then elaborate on atomic event queries and composite event queries, respectively. Language constructs are introduced for both kinds of event queries. The notion of an answer to an event query is then introduced.

2.2.3.1 Essential Traits

For gaining a clear picture of querying event data before going into details regarding XChange event queries as means for querying incoming events, let's take a look at essential traits that event queries have. These traits set XChange event queries apart from Web queries and other existing work on querying events (or event detection). Thus, their brief explanation here is intended to avoid confusions and misunderstanding of event language constructs.

Event Query vs. Web Query Event queries and Web queries serve for different purposes – querying an incoming stream of events vs. querying Web resources; thus, they differ considerably in the communication strategy used, the querying capabilities, and the query processing. Event queries are fed with event data (to be queried) in a push manner, while querying Web resources is done using a pull strategy. Event queries query not only for single events but also for temporal combinations of incoming events, while Web queries lack constructs for dealing with temporal patterns over events. Event queries need to be evaluated in an incremental manner as events arrive in a stream-like manner and are not persistent. As information of Web resources is persistent, such requirements are not posed on the evaluation of Web queries. (Section 2.1.2 has already offered a more elaborated discussion on differences between event queries and Web queries.)

Double Purpose. XChange event queries are dual-purpose: they are aimed for event detection and event data extraction. Event queries detect atomic and composite events (Section 2.2.1) that have occurred on the Web. XChange offers a number of high-level constructs for expressing different kind of event combinations (see Section 2.2.3.3). For extracting pieces of information from incoming events, *variables* are specified in event queries. Data items bound to the variables can be subsequently used for raising events or executing updates.

Logical Variables. Variables are placeholders for the data, in the same way as logic programming variables (or Prova variables – see Appendix B). Variables occurring more than once in an event query act as join variables. *Note* that variables can be also bound to reprsentations of (composite) events, not just to parts of atomic events. This feature is useful when e.g. a detected composite event needs to be sent to other interested Web sites (see Section 2.2.3.3).

Bounded Event Lifespan. For processing XChange event queries (i.e. for detecting atomic or composite events as answers to them), events do not need to be kept for ever in memory. Instead, event data is stored as long as it is needed for answering the event queries posed at a Web site. Moreover, the time for which data on any received event is kept in memory is bounded, i.e. the *event lifespan* is bounded. The notion of event history used in the literature [60, 74, 26] would be misleading in the context of XChange, as event data is not kept forever in memory and event queries do not query events received in the past.

Forward-Looking. XChange event queries are *forward-looking*, i.e. they do not have the ability to look (to query events received) in the past. XChange event queries are capable of querying only events whose representation has been received after the event query has been posed (or registered) at a Web site. This is consistent with the clear cut between volatile data (events) and persistent data (Web resources).

Language Constructs. The language XChange offers several constructs for expressing different combinations of events to react upon.

2.2.3.2 Atomic Event Queries

An *atomic event query* is a query against the representation of a single event. It describes a pattern for a single, incoming event message. An atomic event query specification is an Xcerpt query term with an (optional) *absolute temporal restriction* specification.

Query Terms. The "simplest" XChange event query and, at the same time, the building block for more "complex" event queries (for detecting temporal combinations of events) is an Xcerpt query term. Its purpose, when posed against incoming events, is to detect single occurrences of events. Recall that Xcerpt query terms can be used for querying event data, as event data (i.e. event messages) represent data terms. A short overview of term specifications and query terms follows; a more detailed, full explanation can be found in [64].

An *ordered term specification* (denoted by square brackets [ ]) expresses that the order of subterms is relevant, an *unordered term specification* (denoted by curly braces {}) expresses that the order of subterms is irrelevant. Ordered subterms are needed for text-oriented XML. Unordered subterms are convenient with database-oriented XML. *Total* or *partial* (event and

Web) query patterns can be specified. A query term q using a partial specification (denoted by *double* brackets [[ ]] or braces {{}}) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of q and that (2) might contain further subterms without corresponding subterms in q. In contrast, a query term q using a total specification (denoted by *single* brackets [ ] or braces {}) does not match with terms that contain additional subterms without corresponding subterms in q.

Example 2.4 (Simple XChange Atomic Event Query) *The following XChange atomic event query is a pattern that matches with all incoming events that a Web site receives (recall from Section 2.2.2 that XChange event messages have a root labelled `event` belonging to the `xchange` namespace).*

```
xchange:event {{
 }}
```

Query terms are (possibly incomplete) patterns for the data to be queried. Query patterns can contain variables for extracting pieces of information from data terms (representing event data or data from Web resources). Variables (preceded by the keyword `var`) are placeholders for data. Variable restrictions can also be specified, by writing `var X − > p` (read *as*), which restrict the bindings of the variables to those terms that are matched by the restriction pattern `p`. Query terms may contain querying constructs – comfortable means for query specification – such as:

- *descendant*, for expressing incompleteness in breadth,
- *without*, for expressing subterm negation,
- *except*, for excluding certain subterms from a variable binding,
- *optional*, for specifying optional patterns inside query terms.

These and other constructs are explained in more detail in [64].

Example 2.5 (Variables Inside XChange Atomic Event Queries) *The XChange atomic event query below detects event messages notifying a phone conference. The subject to be discussed and the time at which the phone conference should be held are of interest and thus are to be bound to the variables `S` and `T`, respectively.*

```
xchange:event {{
  xchange:sender {"http://organiser.de/secretary/"},
  phone-conference {{
     subject { var S },
     time { var T }
   }}
 }}
```

For determining answers to atomic event queries and thus bindings for the variables, the event manager of an XChange-aware Web site attempts to match each incoming event received with the currently posed atomic event queries (which themselves may be part of composite event queries). Query terms are matched against event data (or, in case of stantadr queries, data

of Web resources) by means of a novel unification method called Simulation Unification [65, 64], which can handle querying constructs such as partial specifications, optional subterms, or negation of subterms. Informally, a query term q simulation unifies (or simply matches) a data term d if q's structure can be found in d. The outcome of simulation unifying q and d is a set of substitutions for the variables in q. XChange event queries (event part) and Web queries (condition part) are based on query terms and find substitutions for the variables that are then subsequently used in the action part (event raising or transaction specification) of a rule.

Example 2.6 (XChange Event Message Notifying a Phone Conference) *Assume that the organiser of Mrs. Smith uses a rule containing the atomic event query of the previous example. An excerpt of an event message is given in the following using the term syntax:*

```
xchange:event {
  xchange:sender    { "http://organiser.de/secretary/" },
  xchange:recipient { "http://organiser.de/Smith/" },
  xchange:raising-time   { "2005-04-11T10:05:32" },
  xchange:reception-time { "2005-04-11T10:07:02" },
  xchange:id { "1235" },
  phone-conference {
     subject { "Deliverable D5" },
     time { "2005-04-25T14:00" },
     participants {... },
     ...
   }
 }
```

The atomic event query of Example 2.5 detects the above phone conference notification; the evaluation of the atomic event query against the event message results in the following assignments for the variables: $S \mapsto$ `"Deliverable D5"` *and* $T \mapsto$ `"2005-04-25T14:00"`*. Upon reception of other phone conference announcements having the specified pattern, the variables* S *and* T *will be bound to other data items.*

Variables can be used not only *inside* event queries (e.g. variables `S, T` in Example 2.5), but also *outside* event queries. In the latter case, variables are to be bound to the whole event message that matches the atomic event query.

Example 2.7 (Variables Outside XChange Atomic Event Queries) *The following XChange atomic event query is a slight modification of the event query of Example 2.5. Both event queries detect phone conference announcements; the following one binds the variable* `Msg` *to the data term matching the given event pattern.*

```
var Msg -> xchange:event {{
             xchange:sender {"http://organiser.de/secretary/"},
             phone-conference {{ }}
           }}
```

Upon reception of the event message of Example 2.6, the above atomic event query evaluates successfully and binds the variable `Msg` *to the received event message (i.e. the substitution for variable* `Msg` *is exactly the data term of Example 2.6).*

Posing Conditions on Atomic Event Queries. Xcerpt query terms may be further restricted by constraints (e.g. arithmetic expressions on variables occurring in the query term) in a so-called condition box, which has been introduced to source out all restrictions that are not pattern-based. Being Xcerpt query terms, atomic event queries inherit the condition box specification. The keyword `where` (as in Xcerpt) introduces such conditions on atomic event queries.

Example 2.8 (Conditions on XChange Atomic Event Queries) *The following XChange atomic event query detects event messages notifying a flight delay of more than forty-five minutes.*

```
xchange:event {{
  xchange:sender {"http://airline.com"},
  delay-notification {{
     flight-number { var N },
     minutes-delay { var D }
   }}
 }} where { var D > 45}
```

Only events are detected that satisfy the given time constraint. More than one constraint on the variables occurring in the event query can be specified in the `where` *clause. For a more detailed discussion on condition box specification, see Section 4.5.4 of [64].*

Absolute Temporal Restrictions. *Absolute temporal restrictions* are used to restrict the events that are considered relevant for an event query to those that have occurred in the specified time interval. An event occurs in a time interval if the time point at which its representation has been received lies inside the time interval.

XChange absolute time restrictions can be specified by means of a fixed starting and ending point (i.e. a finite time interval) following the keyword `in`. The starting point of such a restricting interval can be implicit (i.e. the time point at which the event query has been registered), in which case it follows the keyword `before`. Thus, an XChange atomic event query specification is defined as:

```
At_EvQ ::= Query_Term
        |  At_EvQ  "in"     Finite_Time_Interval
        |  At_EvQ  "before" Time_Point
```

Note that the production rules defining the nonterminal `Query_Term` are not given here; they can be found in [64].

```
Finite_Time_Interval ::=   "[" Time_Point ".." Time_Point "]"
Time_Point           ::=   ISO_8601_format
```

Time points in XChange are given using the ISO 8601 standard format for representing dates and times.[6] Calendar date, week date, time of the day, and date and time can be represented using the standard. Representations begin with the largest element (e.g. year) followed by smaller elements (e.g. month followed by day). *Note* that years represent the Gregorian

[6] ISO 8601, `http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html`

calendar's years. When a calendar date is followed by the time of the day in the ISO 8601 representation, the capital letter T is used to separate the date and time components. For example, 2005-07-07T14:05:00 represents five minutes after two o'clock in the afternoon of July 7, 2005.

By using the ISO 8601 format for specifying dates and times, it is assumed that all parties use the same calendar – the Gregorian one. In order to facilitate communication between personalised reactive systems whose owners use different calendars, means for defining calendars and reasoning with calendar data are needed. Richer temporal specifications are conceivable in XChange. This can be achieved e.g. by integrating into XChange a calendar and temporal type system such as CaTTS (Calendar and Time Type System) [19].

Example 2.9 (XChange Atomic Event Query Specifying Temporal Restriction) *The following XChange atomic event query detects only such events whose representations have been received before 2005-07-07T14:00:00 and, of course, match the given pattern.*

```
xchange:event {{
  content {{ }}
 }} before 2005-07-07T14:00:00
```

Note that no event is detected if the time interval or the time point specified as temporal restriction for an event query is in the past. The situation is encountered either because of a human error in programming (e.g. writing as year 1005 instead of 2005), or because the time interval for which an event query supposed to detect events has passed.

Example 2.10 (XChange Atomic Event Query Detecting Discounts) *An XChange atomic event query that detects insertion of discounts for flights from Munich to Paris that are received as notifications before July 7, 2005 is given next.*

```
xchange:event {{
  flight {{
    from {"Munich"}, to {"Paris"},
    new-discount { var D }
   }}
 }} before 2005-07-07T10:00:00
```

Note that insertions can be notified by using another structure for event messages. However, the update specification that has been used to perform the insertion can not be sent as content of event messages since update specifications are not data terms. A detailed introduction into XChange update specifications is given in Section 2.4.1.

2.2.3.3 Composite Event Queries

The capability to detect and react to *composite events*, e.g. sequences of events that have occurred possibly at different Web sites within a specified time interval, is needed for many Web-based reactive applications. However, (to the best of our knowledge) existing languages for reactivity on the Web do in general *not* consider the issues of detecting and reacting to such composite events. For processing of *composite events*, XChange offers *composite event queries*.

Composite event queries are specified by means of atomic event queries combined using XChange composite event query constructs. XChange offers a considerable number of such

constructs along two dimensions: *temporal restrictions* and *event compositions*. This section gives an introduction into XChange constructs for composite event queries. Their syntax and informal semantics are given here; a formal definition of the semantics is given in Section 2.2.4.

Temporal Restrictions. The role of temporal restrictions on composite event queries is twofold: they specify interest in events that occur in a given time interval or have a given duration, and ensure that event data can be released after a bounded time (this is realised by using *legal composite event queries*, notion introduced in Section 2.2.3.4).

Note that temporal restrictions do not affect the time point of answer (instance of event query) detection, they only restrict the events that are answer components. Temporal restrictions determine if a rule is fired or not (depending whether the events contained in a candidate answer satisfy the specified time constraint or not) but *when* the rule is fired depends only on the events received.

Absolute Temporal Restrictions. Like for atomic event queries, temporal restrictions can be specified also for composite event queries, posing temporal restrictions on the constituent events.

```
Comp_EvQ ::= Comp_EvQ  "in"      Finite_Time_Interval
          |  Comp_EvQ  "before"  Time_Point
```

Recall that composite events (detected using composite event queries) do not have an occurrence time, like atomic events do. Instead, they have a duration determined by their beginning and ending time, respectively. A composite event `c` is a candidate answer to a composite event query of the form

- `CEQ in [Time_Point`$_1$ `.. Time_Point`$_2$`]` if the beginning time of `c` is greater than or equal to `Time_Point`$_1$ and the ending time of `c` is less than or equal to `Time_Point`$_2$;
- `CEQ before Time_Point` if the ending time of `c` is less than or equal to `Time_Point`. Clearly the above stated temporal conditions on `c` are not enough for detecting `c` as an answer to the composite event query `CEQ`.

Relative Temporal Restrictions. Besides absolute temporal restrictions, also *relative temporal restrictions*, given by a duration, can be specified for composite event queries. This decision is rather straightforward considering that each composite event has a length of time and restricting it may be very useful in practice. For an instance of such a composite event query (i.e. a composite event), the difference between the ending time and the beginning time of the instance needs to be less than or equal to the given duration. Relative temporal restrictions can be given as positive numbers of years, days, hours, minutes, or seconds and their specification follows the keyword `within`.

```
Comp_EvQ ::= Comp_EvQ  "within" Duration
Duration ::= Nr DTime
DTime    ::= "second" | "minute" | "hour" | "day" | "month" | "year"
```

XChange requires every composite event query to be accompanied by a temporal restriction specification. This makes it possible to release each (atomic or partially detected composite)

event (i.e. to release answers to event queries, or partial instantiations of them) at each Web site after a finite time. A detailed discussion on the temporal restrictions that should accompany event query specifications for releasing event data is postponed to Section 2.2.3.4. Thus, language design enforces the requirement of a bounded event lifespan and the clear distinction between persistent vs. volatile data.

Event Composition. This section introduces XChange's constructs for specifying (temporal) patterns over more than one incoming event. With XChange, one can specify *conjunctions*, *temporally ordered conjunctions*, *inclusive disjunctions*, *exclusions*, *occurrences*, and *multiple inclusions and exclusions* of event queries. A discussion of other constructs for event composition that might be useful in practice is given at the end of this section.

Notation. For simplifying the reading and understanding of examples for composite event queries, a short notation for the representations of events is used in this section. Thus, the event messages and the queries against them are 'lifted' by leaving the envelope of event representation out. This means that only the content is specified in event queries *and* incoming events. Consider the following XChange event query and an incoming event message (they give actually a shape for event queries and event messages in XChange):

```
xchange:event {{
    xchange:sender {...},
    content {{ }}
}}
```

```
xchange:event {
   xchange:sender {...},
   xchange:recipient {...},
   xchange:raising-time {...},
   xchange:reception-time {...},
   xchange:id {...},
   content { ... }
}
```

In the remainder of this section, they are written like:

```
content {{ }}
```

```
content { ... }
```

The role of the ids of event messages is taken by a subscript for the content of the event message; however, subscripts are used only when they are needed for differentiating between event messages having the same content. The incoming event stream is a sequence of event messages given in short notation and separated by ','; the direction in which the incoming event stream grows is from left to right (denoted by $--$ ... $-->$).

Also, *beginning_time*(*comp_event*) and *ending_time*(*comp_event*) are used for denoting the beginning and ending time, respectively, of composite event *comp_event*.

Conjunctions. *Conjunctions* specify that instances of each of the specified event queries needs to be detected in order to detect the conjunctive event query. That is, an answer to each of the component event queries needs to be found in order to find an answer to the conjunctive event query. The order in which events occur is not of importance. This is reflected also in the specification of such an event query – by using curly braces.

A conjunctive event query has arity n and at least one event query needs to be specified ($1 \leq n$). Keyword `and` introduces such a composite event query in XChange. The grammar rule defining conjunction event queries in XChange is the following:

```
Comp_EvQ ::= "and" "{" EvQ  ("," EvQ)* "}"
```

Example 2.11 (XChange Event Query Specifying Conjunction (1)) *The following event query specifies interest in the occurrence of pairs of events whose contents match the atomic event queries* a{{}} *and* b{{}}*, respectively:*

```
and {
    a {{ }},
    b {{ }}
 }
```

Assume that the following excerpt of the incoming event stream is received by a Web site after the above event query has been registered:

```
  -- b {c},  g {a,b},  a {d},  a {e},  b {e} -->
```

After receiving b{c}*, one of the atomic event queries has a match and thus a partial instantiation of the whole event query exists. Upon reception of* a{d} *an instance of the event query is detected (i.e. an answer to the event query has been found); the answer has* b{c} *and* a{d} *as components. The beginning time of the answer is the occurrence time of* b{c}*, its ending time is the occurrence time of* a{d}*.*

Upon reception of a{e}*, another answer to the event query is detected having as components* b{c} *and* a{e}*. Upon reception of* b{e}*, the event query has other two answers, one made of* a{d} *and* b{e}*, and one of* a{e} *and* b{e}*.*

Example 2.12 (XChange Event Query Specifying Conjunction (2)) *The following event query is a slight modification of Example 2.11 (above), where the variable X is to be bound to the content of the incoming event messages that match the two atomic event queries.*

```
and {
    a {{ var X }},
    b {{ var X }}
 }
```

Assume that the event stream of the previous example is received by the Web site where the event query is registered. Recall that variables require equality when occurring more than once in an event query (like logic programming variables). Thus, upon reception of b{c} *the event query is partial instantiated and the variable X is bound to c. The reception of* a{d} *offers a match for the atomic event query* a{{ *var* X}} *and the assignment $X \mapsto d$ for the variable, but no instance of the whole event query is detected at this point. An answer to the conjunction event query is detected upon reception of* b{e}*; the answer components are* a{e} *and* b{e}*, and the variable X is bound to e.*

Example 2.13 (XChange Event Query Specifying Conjunction) *Mrs. Smith wants to visit an exhibition of G. Barthouil on a rainy day. The next XChange event query is used to detect the conjunction of the exhibition notification and the desired weather forecast notification that are sent by appropriate Web services.*

```
and {
  xchange:event {{
    xchange:sender {"http://artactif.com"},
```

```
    exhibition {{ painter {"G. Barthouil"},
                  location {"Marseille"},
                  time-interval { var TI }
               }}
   }},
  xchange:event {{
    xchange:sender {"http://weather.com"},
    forecast { date { var D }, city {"Marseille"},
               info {"It's going to rain."} }
   }}
} before 2005-08-16T11:15:00
where var D included-in var TI
```

Temporally Ordered Conjunctions. *Temporally ordered conjunctions* specify that the occurrences of the components need to be successive in terms of time (i.e. query for sequences of events).

The keyword `andthen` introduces such an event query whose component event queries are enclosed in square brackets (for denoting that the order in which events occur is of importance). A temporally ordered conjunction event query has nth arity and at least two event queries need to be specified ($2 \leq n$).

A total specification (i.e. single square brackets) expresses that the answer to such a composite event query contains only the instances of the component event queries. Between the instances of the specified event queries other events might occur that do not match the specified event queries; they neither influence the successful evaluation of the event query, nor are part of the answer.

A partial specification (i.e. double square brackets) for a temporally ordered conjunction event query expresses that the answer contains besides the events that answer the component event queries also all events that have occurred in-between. The practical need for total *and* partial specifications for such event queries has been already motivated by the examples of Section 2.1.7.

The grammar rule for the temporally ordered conjunction event queries are given next:

```
Comp_EvQ ::= "andthen" "[" EvQ  ("," EvQ)+ "]"
          |  "andthen" "[[" EvQ  ("," EvQ)+ "]]"
          |  "andthen" "[[" EvQ  ("," "collect" Query_Term "," EvQ)+ "]
```

For determining answers to temporally ordered conjunction event queries, the temporal order between incoming events needs to be taken into account. An atomic event occurs before an (atomic or composite) event if and only if its occurrence time is before the occurrence or the beginning time of the second event on the time axis of the incoming events. A composite event ce_1 occurs before another composite event ce_2 (i.e. they are successive in terms of time) if and only if the ending time of ce_1 is less than the beginning time of ce_2.

Example 2.14 (Event Query Specifying Temporally Ordered Conjunction (1)) *The following event query specifies interest in the occurrence of sequences of events having content with label a and b, respectively.*

```
andthen [
    a {{ }},
```

```
    b {{ }}
 ]
```

Consider (again) the excerpt of the incoming event stream received by a Web site where the above event query has been registered:

```
  -- b {c},  g {a,b},  a {d},  a {e},  b {e} -->
```

The above event query gets a partial instantiation only upon reception of a{d}, as the event query looks for sequences of events that begin with a-labelled events (or more precisely event contents). Upon reception of a{e} another partial instantiation of the event query is realised. Upon reception of b{e}, two answers to the event query are detected, one represents the sequence a{d}, b{e}, and one the sequence a{e}, b{e}. The fact that other events have been received between the reception of a{d} and b{e} does not affect answering the event query with the sequence of these two events.

Note the difference to the answers of the event query of Example 2.11: sequences b{c}, a{d} and b{c}, a{e} are not answers to the event query as the temporal order between these events is not the desired one.

Example 2.15 (Event Query Specifying Temporally Ordered Conjunction (2)) *The following event query specifies interest not only in sequences of events having content with label a and b, respectively, but also in all events that have occurred in-between.*

```
andthen [[
    a {{ }},
    b {{ }}
 ]]
```

Assume that the above event query is registered at a Web site that receives the excerpt of the event stream used in the previous examples. Upon reception of b{e} two answers to the event query are detected, one represents the sequence a{d}, a{e}, b{e} and one the sequence a{e}, b{e}. The first sequence that is detected as answer to the event query collects the event a{e} because it is received between the answers to the two component, atomic event queries.

Example 2.16 (Event Query Specifying Temporally Ordered Conjunction (3)) *An andthen event query can also be specified to collect only events with a particular pattern.*

```
andthen [[
    a {{ }},
    collect b {{ var X }},
    c {{ }}
 ]]
```

The following excerpt of the event stream received by a Web site where the above event query is posed is used to explain the outcome of such a query:

```
  -- a {e},  b {e},  b {f}, d {}, c {e} -->
```

Upon reception of $c\{e\}$ an answer to the event query is detected, it represents the sequence $a\{e\}$, $b\{e\}$, $b\{f\}$, $c\{e\}$. The bindings obtained for the variable X are $X \mapsto e \vee X \mapsto f$. Note that the occurrence of event $d\{\}$ does not affect the successful evaluation of the event query and is not part of the answer as only non-empty b-labelled events are collected.

Example 2.17 (Event Query Specifying Temporally Ordered Conjunction) *The next XChange event query is used to detect the notification of a flight cancellation and afterwards, within two hours from its reception, the detection of a notification informing that the accommodation is not granted by the airline.*

```
andthen [
  xchange:event {{
    xchange:sender {"http://airline.com"},
    cancellation-notification {{
     flight {{ number { var Number } }} }}
   }},
  xchange:event {{
    xchange:sender {"http://airline.com"},
    important {"Accommodation is not granted!"}
   }}
 ] within 2 hour
```

Inclusive Disjunctions. *Inclusive disjunctions* specify that the occurrence of an instance of any of the specified event queries suffices for detecting the disjunction event query. A reactive rule having as event part an inclusive disjunction event query is fired each time an answer to the specified, component event queries is found. If the component event queries are atomic then the rule is fired each time an event matching one of the atomic event queries is received. Even if a temporal restriction specification accompanies such an event query, the time point of answer detection is not influenced.

The inclusive disjunction event query has arity n and at least one event query needs to be specified ($1 \leq n$). The keyword `or` denotes an inclusive disjunction in XChange and the event queries are enclosed in curly braces.

```
Comp_EvQ ::= "or" "{" EvQ  ("," EvQ)* "}"
```

Note that exclusive disjunctions of event queries can also be specified in XChange by means of the multiple inclusion and exclusion event queries. They specify a generalised exclusive disjunction of event queries and are discussed later in this section.

Example 2.18 (XChange Event Query Specifying Inclusive Disjunction (1)) *The following event query specifies interest in the occurrence of events having content with label a or b.*

```
or {
    a {{ var X }},
    b {{ var X }}
 }
```

Consider the following excerpt of the event stream received by a Web site where the event query is registered:

```
-- b {c},   g {a,b},   a {d} -->
```

Upon reception of $b\{c\}$ *an answer to the event query is detected. The variable* X *gets a binding,* $X \mapsto c$*. Upon reception of* $g\{a, b\}$ *nothing happens as none of the specified atomic event queries matches it. The reception of* $a\{d\}$ *leads to a new answer for the inclusive disjunction event query and a new binding for the variable,* $X \mapsto d$*. Thus, each time an event matching one of the two atomic event queries is received a new answer to the inclusive disjunction event query is detected and a new binding for the variable is found.*

Example 2.19 (XChange Event Query Specifying Inclusive Disjunction) *After Orange, Mrs. Smith wants to visit Arles and Nîmes. The next city to visit is chosen depending on the notification of train tickets and hotel reservation made by appropriate services.*

```
or {
  xchange:event {{
    xchange:sender {"http://service-nimes.fr"},
    service-notification {{
      train {{  date {"2005-08-10"},
                from {"Orange"}, to {"Nimes"}  }},
      hotel {{ }}
    }}
   }},
  xchange:event {{
    xchange:sender {"http://reservations-arles.fr"},
    reservation-notification {{
      train {{ date {"2005-08-10"},
               from {"Orange"}, to {"Arles"} }},
      accommodation {{ }}
     }}
   }}
 } before 2005-05-02T21:30:00
```

Exclusions. *Exclusions* (event negation) specify that no instance of the given event query should have occurred in a finite time interval in order to detect the exclusion event query.

A finite time interval acting as a monitoring window over the incoming event stream is necessary for the detection of non-occurrence of an event. After an exclusion event query is posed at a Web site, the incoming events are queried for determining whether an instance of the specified event query occurred or not. If an instance of the event query occurs then the exclusion event query has no successful evaluation. At a point in time, it can be determined whether an event query instance has occurred or not, but one can not predict what kind of events the future will bring. Thus, the event manager needs to know the time point until non-occurrence (or occurrence) of event query instances is to be monitored. A (monitoring) time interval for exclusion event queries is given by a finite time interval or by a composite event query (recall that their instances have a beginning and an ending time and thus determine a time interval).

The keyword `without` introduces exclusion event queries in XChange and the finite time interval specification or the composite event query follows the keyword `during`. The following grammar rules define the exclusion event queries:

```
Comp_EvQ ::= "without"   "{" EvQ "}"   "during"   "{" Comp_EvQ "}"
           | "without"   "{" EvQ "}"   "during"   Finite_Time_Interval
```

An XChange exclusion event query is evaluated at the end of the monitoring time interval. That is, the non-occurrence of an event query instance is evaluated at each successful evaluation of the composite event query or at the end of the given finite time interval. The firing time point of a reactive rule having as event part an exclusion event query is the ending time of an instance of the composite event query or the ending time of the finite time interval. Recall that the firing time point of a reactive rule having as event part an event query of the form *Comp_EvQ in Finite_Time_Interval* (absolute temporal restriction on a composite event query) is the ending time of a detected instance of the composite event query *Comp_EvQ*. To reflect the difference between the time point of evaluation and thus the firing time point of an associated rule, the keyword `during` is used instead of `in` for exclusion event queries.

Example 2.20 (XChange Event Query Specifying Exclusion (1)) *The following event query specifies interest in the non-occurrence of c-labelled events during occurrence of sequences of events labelled a and b, respectively.*

```
without {
    c {{ }}
  } during {
            andthen [ a {{ }}, b {{ }} ]
           }
```

Assume that the Web site where the above exclusion event query is registered receives the following excerpt of the incoming event stream:

```
  -- a {e},   e {f},   a {d},   c {e, f{g} },   b {f} -->
```

*After receiving the event $a\{e\}$, occurrences of events matching $c\{\{\}\}$ or $b\{\{\}\}$ are monitored. Upon reception of $b\{f\}$, the **andthen** event query is successfully evaluated with two sequences as answers, one made of $a\{e\}$ and $b\{f\}$, and one of $a\{d\}$ and $b\{f\}$. Though, the evaluation of the whole event query is not successful (i.e. the event query has no answer) as the event $c\{e, f\{g\}\}$ has occurred during both answers to the **andthen** event query ($c\{\{\}\} \preceq c\{e, f\{g\}\}$ and an answer is found to the event query whose exclusion is of interest).*

Example 2.21 (XChange Event Query Specifying Exclusion (2)) *The following exclusion event query is a slight modification of the previous example where some of the component atomic event queries are augmented with variables.*

```
without {
    c {{ var X }}
  } during {
            andthen [ a {{ var X }}, b {{ }} ]
           }
```

*Consider that the above event query is to be evaluated against the excerpt of the incoming event stream given in the previous example. Upon reception of $a\{e\}$ the **andthen** event query is partially evaluated and the variable gets a binding, the substitution $\sigma_1 = \{X \mapsto e\}$ is obtained.*

The reception of $e\{f\}$ *does not influence the evaluation of the event query. Upon reception of* $a\{d\}$ *another instance of the* **andthen** *event query is partially evaluated and a possible binding for the variable is found, the substitution* $\sigma_2 = \{X \mapsto d\}$ *is obtained. The event* $c\{e, f\{g\}\}$ *matches the event query whose exclusion is of interest; the result of* $c\{\{var\ X\}\} \preceq c\{e, f\{g\}\}$ *gives two possible bindings for the variable,* $\Sigma_3 = \{\{X \mapsto e\}, \{X \mapsto f\{g\}\}\}$*. Receiving* $b\{f\}$ *determines two answers to be found for the* **andthen** *event query. Now, the exclusion event query can be evaluated. Recall that variables used in event queries require equality when occurring more than once in a query. Thus, the whole event query is evaluated successfully only once against the given incoming event stream, with the sequence* $a\{d\}$, $b\{f\}$ *(no c-labelled events having a d-child are received, thus the event query is successful). The variable substitution obtained is* $\sigma = \{X \mapsto d\}$ *(*$\sigma_2 \wedge \neg\Sigma_3$*).*

Example 2.22 (XChange Event Query Specifying Exclusion (3)) *The following exclusion event query detects the non-occurrence of c-labelled events within the given time interval.*

```
without {
    c {{ }}
  } during [2005-05-22T14:00:00 .. 2005-05-22T20:00:00]
```

Assume that no c-labelled events have occurred within the specified finite time interval; at time point 2005-05-22T20:00:00 *the exclusion event query evaluates successfully.*

Variables occurring in the event queries whose exclusion is of interest (i.e. event queries specified after the keyword `without`) need to have at least one *defining* occurrence in the (whole) event query in order to be further used in an event query or other parts of XChange rules. Each variable occurrence in XChange rules is associated with a *polarity* for determining whether a variable occurring in the event part (or condition part) of the rule can be used in the condition and action part (or, just in the action part, respectively) of the rule or not (i.e. determining rules' *range restriction*). A *negative* polarity of a variable occurrence expresses a defining occurrence of the variable. A *positive* polarity expresses a non-defining variable occurrence. The polarity of Xcerpt query terms (defined in [64]) is extended for XChange event queries. The polarity of event queries and the range restriction of XChange rules are postponed to Section 2.5.5. In Example 2.21 the first occurrence of the variable X has positive polarity as it occurs inside the event query whose exclusion is of interest, the second occurrence of X has a negative polarity. Thus, the variable can be used outside the event query (e.g. in complex event queries having as one of the components the exclusion event query).

Example 2.23 (XChange Event Query Specifying Event Exclusion) *The event query below detects if the notifications of two online reservations made on 10th of July 2005 are not received within ten days.*

```
without {
  and {
      xchange:event {{
              flight-reservation-notification {{ }}
       }},
      xchange:event {{
              hotel-reservation-notification {{ }}
       }}
```

```
  }
 } during [2005-07-10..2005-07-20]
```

Quantifications. Quantifications in event queries are used to detect instances that occur (at least, at most, or exactly) a number of times in a given time interval or between occurrences of other event query instances.

The keyword `times` introduces such quantification event queries in XChange. The occurrences of instances of a given event query (EvQ) are to be counted within a time interval, which is either determined by instances of a given composite event query ($Comp_EvQ$) or is directly given as a finite time interval specification ($Finite_Time_Interval$). The following grammar rules define such composite event queries in XChange:

```
Comp_EvQ ::=
    "times" M ("any" Vars)?  "{" EvQ "}"   "during"   "{" Comp_EvQ "}"
 |  "times" M ("any" Vars)?  "{" EvQ "}"   "during"   Finite_Time_Interval

M       ::= ("atleast" | "atmost")? Nr Nr      ::= [1-9][0-9]*
Vars    ::= "var" Var_Name ("," "var" Var_Name)
```

Example 2.24 (XChange Event Query Specifying Quantification (1)) *The following event query specifies interest in the reception of at least three messages from the secretary within the specified time interval. Also, the subjects of the messages are of interest (e.g. for using them in the action part of the rule having the following event part).*

```
times atleast 3 {
    secretary-message {{
      subject { var S },
      content {{ }}
    }}
  } during [2005-05-23T08:00..2005-05-23T18:00 ]
```

The event query evaluates successfully if between 2005-05-23T08:00 and 2005-05-23T18:00 at least three messages are received having the same subject. Being a logical variable, the variable S requires equality. By dropping the variable S (i.e. specifying just subject{{}} instead of subject{var s}), the event query detects the reception of at least three messages with possibly different subjects, but these subjects can not be further used, as no variable is bound to this information.

By means of the constructs introduced so far, one can detect situations like the reception of three messages with *different* subjects, but one can not react upon them by e.g. sending a response message containing a list of all three messages' subjects. To overcome this, the approach taken consists in introducing *existential quantified variables*, i.e. variables that do not require equality of bindings in selecting data items. Informally, existential quantification expresses that at least one binding for the given variable exists.

The specification of existentially quantified variables follows the occurrence specification (`times M`). The keyword `any` precedes the list of the existential variables used in an event query. Variables not declared as existentially quantified require equality when occurring more than once in an event query. Declaring a variable as existentially quantified in an event query

applies to all its occurrences in the component event queries (the property of being existentially quantified for a variable is inherited in a top down manner to component event queries).

Example 2.25 (Event Query Specifying Quantification (2)) *The following event query specifies interest in the reception of at least three messages from the secretary within the specified time interval.*

```
times atleast 3 any var S {
    secretary-message {{
      subject { var S },
      content {{ }}
    }}
  } during [2005-05-23T08:00..2005-05-23T18:00 ]
```

Assume that the following three messages are received within the given time interval (other kinds of events might have also been received, but their occurrence does not influence the evaluation of the above event query):

```
secretary-message {        secretary-message {        secretary-message {
      subject {"WG I1"},         subject {"WG I5"},         subject {"TTA"},
      content {...},             content {...},             content {...},
      ...                        ...                        ...
}                          }                          }
```

The quantification event query evaluates successfully against an incoming event stream containing the above messages (whose reception times lie inside the given time interval) and the variable S has three possible bindings; the substitution set $\Sigma = \{\{S \mapsto "WG\ I1"\}, \{S \mapsto "WG\ I5"\}, \{S \mapsto "TTA"\}\}$ *is obtained.*

Multiple Inclusions and Exclusions. *Multiple inclusions and exclusions* detect occurrences of a given number of event query instances and the non-occurrence of instances of the other specified event queries. It expresses a generalised exclusive disjunction of event queries.

The keyword `of` preceded by an occurrence specification (e.g. atleast 2) introduces such event queries in XChange. The occurrence specification expresses how many of the specified event queries need to have instances; the whole event query evaluates successfully if instances of the other event queries do not occur during a finite time interval. Again, such a time interval can be given through a composite event query or directly by giving its begin and end time points. The multiple inclusions and exclusions event query is used for detecting occurrences of some event queries and non-occurrences (exclusion) of others, thus it can be evaluated just at the end of evaluation of the given composite event query or at the end of the time interval (the keyword `during` is used). Existentially quantified variables (i.e. variables that do not require equality when occurring more than once in an event query) can be used also in multiple inclusions and exclusions event queries. At least one event query needs to be specified after the occurrence specification. The following grammar rules define such event queries in XChange:

```
Comp_EvQ ::=
   M "of" ("any" Vars)? "{" EvQ  ("," EvQ)* "}" "during" "{" Comp_EvQ "}"
 | M "of" ("any" Vars)? "{" EvQ  ("," EvQ)* "}"
                          "during" Finite_Time_Interval
```

Recall that `M` is of the form `atleast Nr`, `atmost Nr`, or just `Nr` (it has been introduced at the `times` construct for event queries). `Nr` is a positive integer greater than `0` and less than the number of event queries specified after keyword `of`. An event query of the form 1 *of* $\{EvQ_1, EvQ_2, ..., EvQ_n\}$ (where $Nr = 1$) expresses exclusive disjunction of the instances of the specified event queries.

Example 2.26 (Event Query Specifying Multiple Inclusions and Exclusions (1)) *The following event query specifies an exclusive disjunction of a-labelled and b-labelled events. The occurrence of such events is of interest during instances of a conjunctive event query.*

```
1 of {
  a {{ var X }},
  b {{ var Y }}
 } during {
           and {
            c {{ }},
            d {{ var Y }} }
          }
```

Assume that after registering the event query at a Web site, the following excerpt of the event stream is received:

```
-- d {g,h},   e {d},   a {k,f},   c{e},
c{f,g},   b{f},   d{e}   -->
```

The event query evaluates successfully two times on the above given stream of events; the instances of the event query are composed of $d\{g, h\}$, $a\{k, f\}$, *and* $c\{e\}$ *(for the first answer), and* $d\{g, h\}$, $a\{k, f\}$, *and* $c\{f, g\}$ *(for the second answer). The sequences composed of* $c\{e\}$, $b\{f\}$, *and* $d\{e\}$, *and* $c\{f, g\}$, $b\{f\}$, *and* $d\{e\}$, *respectively, do not represent answers to the whole event query as the variable* Y *is a logical one and requires equality.*

Multiple inclusions and exclusions event queries specify interest in occurrence of event query instances and non-occurrence (event exclusion) of other event query instances; one does not know beforehand which event queries will be answered and which not. Thus, it might be the case that not all variables used in the event queries will have bindings. Recall the discussion on variable substitutions for the case of exclusion event queries. Similarly, variables occurring in the event queries whose inclusion or exclusion is of interest (i.e. event queries specified after the keyword `of`) need to have at least one *defining* occurrence in the (whole) event query in order to be further used in an event query or other parts of XChange rules. In Example 2.26 the variable Y has a defining occurrence as it will be bound when evaluating the conjunctive event query (which determines the monitoring time interval). The occurrence of variable X is non-defining and thus it cannot be further used in the rule having as event part the example event query. (Though, if the event query is part of a more complex event query, the variable X might have a defining occurrence in other parts of the complex event query.)

One might argue that the language is too restrictive because of the requirement of at least one defining occurrence for the variables that need to be used in other parts of an XChange rule. Another approach would consist in introducing a kind of optionality specification for the variables that do not have a defining occurrence in the event query; the specification accompanies these variables in the condition and action part of XChange rules. A default value for the

variable marked "optional" needs also be specified; this value is to be used when no binding for the variables resulted from a successful evaluation of the event query.

Variables Inside and Outside XChange Event Queries. As the examples introduced until now have shown, variables can be bound to *data items* of events received (by using variables inside atomic event queries), or to *atomic events* that have occurred on the Web (by using variables outside atomic event queries). Variables can also be bound to *composite events*, i.e. to answers to composite event queries. This is achieved by using variables outside composite event queries, like

```
var CE → Comp_EvQ
```

The variable CE is to be bound to the answers found for $Comp_EvQ$. Answers to composite event queries contain all atomic events that are used for answering the event query, they are sequences of atomic events whose representation is an XML document[7]. Thus, variables occurring in (more precisely, inside or outside) XChange event queries are bound to data terms.

Nesting XChange Event Queries. XChange constructs for composite event queries can be nested arbitrarily; thus, complex reactive applications can be easily and elegantly implemented in XChange.

Example 2.27 (Nesting XChange Event Queries) *The following example gives a composite event query for detecting occurrences of a flight cancellation, where the airline does not grant an accommodation. For this purpose, a temporal ordered conjunction construct, the exclusion construct and temporal restrictions are combined.*

```
andthen [
    xchange:event {{
      xchange:sender { "http://airline.com" },
      cancellation-notification {{
          flight {{ number { "AI2021" }, date { "2005-08-21" }  }}
          }}
     }},
    without { xchange:event {{
              xchange:sender { "http://airline.com" },
              accommodation-granted {{ hotel {{ }} }}  }}
            } during [2005-08-21T17:00..2005-08-21T19:00]
] within 2 hours
```

Conditions on Composite Event Queries. As for atomic event queries, variables occurring in composite event queries can also be constrained with conditions specified in a `where` clause. (Recall that only non-structural conditions are to be specified, structural conditions are given through event query patterns.) The following grammar rule defines composite event queries with condition box specification (for a detailed explanation of `Condition`, see Section 4.5.4 of [64]):

[7]A more detailed discussion on answers to XChange event queries and their representations is given in Section 2.2.3.5.

```
Comp_EvQ ::= Comp_EvQ "where" "{" Condition ("," Condition)* "}"
```

Example 2.28 (Conditions on XChange Composite Event Queries) *Consider the simple composite event query:*

```
without {
         a { var X }
 } during {
           and {
               b { var X },
               c { var Y }  }
          } where { X < Y }
```

The event query detects conjunctions of b-labelled and c-labelled events with no a-labelled event in-between whose content is the same as for the b event and less than the content of the c event.

It is not that clear which kind of constructs should necessarily be included into a reactive language developed not only for a single kind of applications, but trying to cover different classes of applications. Developing use cases for a language entails introduction of new language constructs and (perhaps) removing others; it also reveals the limits of a language. Moreover, a tradeoff between the expressive power of the language and the ease of its usage needs to be found in designing a language. The design of XChange event queries (and in fact of the whole language XChange) aimed at introducing powerful constructs that ease the programming task.

2.2.3.4 Legal Event Queries

Recall the statement *"XChange event queries are such that volatile data remains volatile"* given in Section 2.1.8. An essential trait of event queries (cf. Section 2.2.3.1) is that they ensure that data of no event is kept forever in memory; that is, the event lifespan is bounded. Though, (composite) event queries as introduced in the previous section can need an unbounded lifespan for events.

Example 2.29 ("Illegal" XChange Composite Event Query) *The composite event query below specifies interest in a-labelled events followed by b-labelled events.*

```
andthen [
    a {{ }},
    b {{ }}
]
```

Consider the excerpt of the incoming stream received at a Web site where the above event query is registered:

```
-- a {e},  b {f},  c {g},  b {h},
a {k} -->
```

XChange assumes no consumption of events, thus the same event may be part of more than one instance of a composite event query. After detecting the instance of the above `andthen` *event query composed of* $a\{e\}$ *and* $b\{f\}$*, the atomic event* $a\{e\}$ *needs to be kept in memory for*

waiting to other b events to occur. Upon reception of $b\{h\}$, using the events kept in memory, another instance of the event query is detected (the instance composed of $a\{e\}$ and $b\{f\}$). The next event received matches the atomic event query $a\{\{\}\}$, thus needs to be kept in memory for b-labelled events that will possibly be received in the future. (If instead of an `andthen` *event query an* `and` *event query is used, besides a-labelled events also b-labelled events need to be kept in memory.) As the event manager can not predict which kind of events will be received, event data needs to be kept forever in memory. For avoiding this, restrictions on composite event queries are posed.*

XChange (composite) event queries are restricted to so-called *legal event queries* that can be evaluated with bounded event lifespan. A formal proof of this statement is given in Chapter 6 of [30]. Legal event queries have the promised trait of keeping the clear cut between persistent and volatile data.

XChange atomic event queries (with or without temporal restrictions) do not require events to be kept in memory; thus, each atomic event query is considered legal. Restrictions are posed only on composite event queries whose answer detection requires partially detected instances to be kept in memory. The main idea is to restrict the time period of monitoring events (which are possible candidates to answering an event query) to a finite time interval (the programmer should specify). The following composite event queries are legal:

- composite event queries with absolute or relative temporal restriction;
- composite event queries specifying exclusions, quantifications, last instance, and multiple inclusions and exclusions of event queries where the monitoring time period is given by a finite time interval (a *during Finite_Time_Interval* specification).

```
Legal_EvQ  ::= At_EvQ
             | LC_EvQ

LC_EvQ ::= Comp_EvQ  "in"      Finite_Time_Interval
       | Comp_EvQ  "before"  Time_Point
       | Comp_EvQ  "within"  Duration
       | "without" "{" EvQ "}" "during" Finite_Time_Interval
       | "times" M ("any" Vars)? "{" EvQ "}" "during" Finite_Time_Interval
       | "last" "{" EvQ "}"  "during"  Finite_Time_Interval
       | M "of" ("any" Vars)? "{" EvQ ("," EvQ)* "}" "during" Finite_Time_Interval
```

The above-mentioned kinds of legal composite event queries are very restrictive; there are other composite event queries that do not belong to the classes mentioned but need only events of bounded lifespan for their evaluation. However, the restrictions offer a set of simple and clear rules to follow for programming legal event queries.

Example 2.30 (XChange Composite Event Query) *The following composite event query is not legal with respect to the definition above (T2 represents a time point.)*

```
andthen [
    a {{ }},
    b {{ }} before T2
]
```

Though, it can be rewritten as a legal composite event query:

```
andthen [
    a {{ }},
    b {{ }} before T2
] before T2
```

Based on the semantics of the event query and the "legality" of its component event queries, one can infer whether the whole composite event query is legal or not. The rules for legal composite event queries need to be extended with the following ones defining (inferred) legal event queries (the first rule needs to be added to the definition of legal composite event queries given above):

```
LC_EvQ  ::= Inf_EvQ

Inf_EvQ ::= "or" "{" Legal_EvQ  ("," Legal_EvQ)* "}"
        | "without" "{" EvQ "}" "during"  LC_EvQ
        | "times" M ("any" Vars)? "{" EvQ "}" "during"  LC_EvQ
        | "last" "{" EvQ "}"  "during" LC_EvQ
        | M "of" ("any" Vars)? "{" EvQ ("," EvQ)* "}" "during" LC_EvQ
```

2.2.3.5 Answers to Event Queries

An answer to an XChange (atomic or composite) event query consists of an (atomic or composite) *event* and a *substitution set*. Answering an XChange event query results in all atomic events that have been used for this purpose and the set of substitutions for the variables occurring in the event query. One of the recognised design principles for Web query languages is the *answer closedness*, meaning that answers to Web queries can be further queried with the Web query language. Considering (sequences of) atomic events answering event queries is rather natural having this principle in mind – answers to event queries can be further queried by event queries.

Considering just the atomic events answering event queries is not enough, the substitution sets of answers to event queries play an important role – they represent communication means between the components of XChange reactive rules: The substitutions for the variables occurring in an event query restrict the possible substitutions for the variables occurring in the Web query and action specification of the same XChange reactive rule. The substitutions provide data for performing the desired actions, for constructing notifications to be sent to other Web sites and for constructing new data to be inserted into (local or remote) Web resources' data. The *maximal* substitution set for *all* variables having at least one defining occurrence in the event query is considered for the answer to an XChange event query. Substitutions for *all* variables are of interest so as to be able to group (e.g. by using the grouping construct `all`) the substitutions when used in the action part of XChange rules. Intuitively, a substitution set Σ answering the event query EvQ is *maximal*, if there exists no substitution set Υ answering EvQ such that Σ is a proper subset of Υ. A more general and formal definition of maximal substitution sets can be found in [64] (Section 7.3, Definition 7.1 on page 147).

Answers to Atomic Event Queries.
An answer to an atomic event query consists of

(i) the *atomic event* whose representation (as event message) matched the event query (and occurred in the given time interval, if a temporal restriction has been specified). Atomic events, are represented as XML documents – they are XChange event messages. A DTD for the representation is given in Section 2.2.2.

(ii) the (maximal) *substitution set* for all variables with a defining occurrence in the event query; the substitutions are the result of matching (*simulation unifying*) the atomic event query with the atomic event given in part (i) of the answer.

Answers to Composite Event Queries.
An *XChange composite event* is a sequence of atomic events that altogether are used for answering a composite event query. Thus, an answer to a composite event query consists of

(i) a *sequence of atomic events* that have occurred and have been used to answer the composite event query.

(ii) the (maximal) *substitution set* for all variables with at least one defining occurrence in the event query; the substitutions are the result of matching the component atomic event queries with the atomic events the answer contains. Clearly, the component atomic events satisfy the temporal pattern given by the composite event query they answer.

Composite events are also represented as XML documents (allowing for further processing) as a (flat) sequence (with an artificial root to make them valid XML) of all atomic events that were used for answering the composite event query. The atomic events are ordered by their reception times. The first and last child elements of a composite event's representation are its beginning time and ending time, respectively. A DTD for the representation of XChange composite events is given next. `xchange:event` represents an XChange event message; a DTD for it is given in Section 2.2.2.

```
<!DOCTYPE xchange:event-seq [
   <!ELEMENT xchange:event-seq (
                      xchange:beginning-time,
                     (xchange:event)*,
                      xchange:ending-time)>

   <!ATTLIST xchange:event xmlns:xchange CDATA #FIXED
             "http://xcerpt.org/xchange">

   <!ELEMENT xchange:beginning-time    (#PCDATA)>
   <!ELEMENT xchange:ending-time       (#PCDATA)>
]>
```

Variables can be bound to composite events, or more exactly to their representation as XML documents. This can be achieved by restricting a variable to the answers to a composite event query – by writing $var\ A \rightarrow Comp_EvQ$. The bindings for variable A are the composite events answering the composite event query $Comp_EvQ$.

Example 2.31 (XML Representation of a Composite Event) *The example shows an answer to the composite event query specifying quantifications; the event query has been given as Example 2.25.*

```
<xchange:event-seq>
    <xchange:beginning-time> 2005-05-23T13:01 </xchange:beginning-time>
    <xchange:event>
        <xchange:sender>    http://lmu.de/secretary </xchange:sender>
        <xchange:recipient> http://lmu.de/smith     </xchange:recipient>
        <xchange:raising-time>   2005-05-23T13:00 </xchange:reception-time>
        <xchange:reception-time> 2005-05-23T13:01 </xchange:reception-time>
        <xchange:reception-id>   42 </xchange:reception-id>
        <secretary-message>
            <subject> Urgent call</subject>
            <content> Werner called regarding ...</text>
        </secretary-message>
    </xchange:event>
    <xchange:event>
        <xchange:sender>    http://lmu.de/secretary </xchange:sender>
        ...
    </xchange:event>
    <xchange:event>
        ...
        <xchange:reception-time> 2005-05-23T15:16 </xchange:reception-time>
        ...
    </xchange:event>
    <xchange:ending-time> 2005-05-23T15:16 </xchange:ending-time>
</xchange:event-seq>
```

Answers to XChange composite event queries (more precisely, their representation) can be "put in an envelope" and sent as event messages to one or more Web sites. Just as it is easy to exchange and query information about atomic events, it is also easy to exchange and query information about composite events. Both kinds of events are data terms (term representation of XML documents), thus Simulation Unification can be applied for further querying (atomic or composite) events.

On Representing the Notion of Answer to Composite Event Queries (Discussion) Other approaches for representing answers have also been investigated, e.g., XML representations mirroring the nested structure of a composite event query. Following the approach, in XChange a composite event is an XML document containing answers to the component event queries and the temporal relations between these answers.

Example 2.32 (XML Representation of a Composite Event (Possible Approach)) *Consider the following composite event query, where $T1$ denotes a specification of a time point:*

```
andthen [
    a {{ }},
    and {
       b {{ }},
       c {{ }}
    }
] before T1
```

Answers to an `andthen` *event query are represented as XML documents with root labelled* `xchange:event-andthen` *containing the answers to the component event queries. Answers to the* `and` *event query are represented as* `xchange:event-and`*-labelled documents. An attribute* `ordered` *denotes ordered and unordered child elements. The answers to the above event query look like in the following (recall that the short notation for atomic events is used, where the root and the parameters of event messages are missing):*

```
<xchange:event-andthen ordered="true">
        <a> .... </a>
        <xchange:event-and ordered="false">
                <c> .... </c>
                <b> .... </b>
        </xchange:event-and>
</xchange:event-andthen>
```

An advantage of having answers mirroring the structure of composite event queries is that Web sites receiving composite events data can very easily determine the whole or the components of the composite event queries they answered. A disadvantage of this approach is that the same atomic events might be found several times in the representation of a single composite event. However, a flat sequence for representing composite events is better. It is simpler and more intuitive for users, since no knowledge of the query structure is required. It leads to an easier definition of declarative semantics (presented in Section 2.2.4), due to the similarity between sets and sequences. Finally, it is desirable for a query language to have similar input and output — and the input of event queries is a collection of atomic events arriving sequentially. In principle, this allows for using the answer to a composite event query as the input to another event query.

2.2.4 Declarative Semantics for Event Queries

Having discussed the language constructs for event queries informally and intuitively in the previous section, we now provide a formal, declarative semantics. A formal semantics is desirable for several reasons [69, chapter 8]. In the context of event queries in XChange, the most compelling are:

- **Standardization.** Formal semantics are useful in the standardization of programming languages. In contrast to informal specifications, formal semantics are clear and unambiguous.

- **Reference for implementors.** Consequently, the availability of formal semantics to language implementors averts misinterpretations that could lead to different and incompatible dialects of the same language in different implementations. A concrete example here is the need for duplicate elimination (cf. Section 2.2.5.3) in the incremental event query evaluation we describe later; without work on formal semantics, duplicate elimination might easily have been overlooked.

- **Reference for users.** Formal semantics are usually concise and can serve as good reference for users of a language to look up the meaning of a particular language construct.

- **Basis for formal proofs.** Formal proofs about programs in a language or about properties of a language are only possible with formal semantics. For example, on grounds of the formal semantics for XChange event queries, one can prove that for every legal event query, there is some upper bound on the life-spans of all events needed to evaluate (see [30]).

- **Better understanding of the language design.** Finally, and maybe most importantly, definition of formal semantics give new insights into the design of a language. A language construct that is hard to define formally is likely to be hard to understand and use for a programmer. Also, the formal definitions help to identify missing or superfluous language constructs.

The definition of declarative semantics for event queries is done in three steps: First, atomic events and the stream of incoming atomic events are defined formally. Next, answers to atomic and composite event queries are defined as sub-sequences of the event stream together with a substitution set for the variables. Finally, an answering-relation $\triangleleft$ is defined between queries and answers. This answering-relation provides the information on *when* a certain query succeeds and *what* its answer is. It is the heart of the declarative semantics and defined by structural induction on the shape of a query.

The definition of the semantics requires some specific notation, which is introduced as we proceed.

2.2.4.1 Atomic Events and Event Stream

Atomic events. Atomic events a are data terms d that are received by the query processor running at some XChange-aware Web site (i.e., they "occur" or "happen") at a reception-time r. Together we write this as $a = d^r$.

The domain of data terms $\mathcal{T}^d$ is defined in [64]. For the understanding of this chapter, only little knowledge about data terms, simulation unification, etc., is required. We do not give a full introduction here and will provide the necessary information along the way.

Reception-time, other time points, and time differences are interpreted in a time domain $(\mathbb{T}, \mathbb{D})$. Throughout this chapter we ignore the syntactical representation of time points and time differences and assume that any time point objects $r, b, b', e, e' \in \mathbb{T}$, and any time difference objects $w \in \mathbb{D}$ are already interpreted objects. While this is a little imprecise on the formal side, it enhances readability of the formal semantics greatly.

The time domain $(\mathbb{T}, \mathbb{D})$ must accommodate time points and time differences (lengths of time). For this work, it must satisfy the following conditions:

- An equality relation $=$ for both time points and time differences is available.

- There is a total order $<$ on time points indicating that some time point lies temporally *before* some other time point.

- In this order, there is a smallest time point 0, and no greatest time point, i.e., time is infinite for the future.

- A minimum min and a maximum max function for time points are available.

- The time difference between two time points $t_1 < t_2$ is $t_2 - t_1$.

- Time differences w, w' can be compared with $w < w'$, indicating that w is shorter than w'.

- Adding a length of time w to a time point t_1 results in a time point $t_2 = t_1 + w$ such that $t_2 - t_1 = w$.

One possibility for the time domain is to use natural numbers $\mathbb{N}$ for both time points and time differences (i.e., $\mathbb{T} = \mathbb{D} = \mathbb{N}$) with the usual $=$, $<$, 0, $+$, $-$. Rational numbers $\mathbb{Q}$ with the usual operators work equally well.

Event stream. All atomic events received form together a *stream of incoming events* (*event stream* for short) on which a query is evaluated. In related work [39, 25], this stream of incoming events is also often called *event history.* A query registered at an XChange-aware site, can only "see" events happening after the query was registered; it cannot look into the "past".[8] This is a basic design assumption that allows to discard each incoming event at some point of time and avoids storing incoming events forever (cf. 2.2.3.1).

The stream of incoming events $\mathcal{E}$ is a finite sequence $\langle a_1, a_2, \dots a_n \rangle_b^e$ of all atomic events $a_i = d_i^{r_i}$ happening in the time interval $[b..e]$. The end of the time interval (the "higher" number) is written in the superscript (the "higher" position) of the sequence, while the begin is written in the subscript. For correct semantics of an event query, the stream of incoming events begins at the time b the event query was registered to the system.

The atomic events of an event stream must all lie inside the interval $[b..e]$, and also be ordered totally with respect to their occurrence time: $b \leq r_1 < \dots < r_n \leq e$. The total ordering of atomic events corresponds to the assumption that no two atomic events happen simultaneously. Note that it suffices to consider a finite sequence: for the evaluation of a query at some point in time, all events needed in the evaluation have a lower temporal bound (the time when the query was registered to the system) and an upper temporal bound (the time when the event is currently evaluated).

To avoid superscript notation, define a function $rcp(a) = r$ for the reception time of an atomic event $a = d^r$.

2.2.4.2 Answers to Event Queries

Answers. The notion is twofold. An answer to some query consists of:

- a sequence of atomic events s that allowed a successful evaluation of the query on the one hand, and

- a set of variable substitutions Σ on the other hand.

This corresponds to the notion of answer in other languages, e.g., Xcerpt, where an answer to a query also consists of matching data terms and substitution sets.

We write answers as a tuple of the event sequence s and the substitution set Σ: (s, Σ). Sometimes the surrounding parenthesis will be dropped to ease readability: s, Σ.

[8] Hence, we prefer the term "stream" in this work, as "history" might lead to the conclusion that an event query looks into the past indefinitely. The term "stream" also integrates nicely with the incremental forward chaining evaluation of event queries provided in 2.2.5

Substitution sets. A substitution set Σ is a (finite) set of substitutions $\sigma_1, \ldots \sigma_n$. A substitution σ assigns values, which are data terms, to variable names. We write $\sigma = \{X \mapsto f[\text{"}42\text{"}], Y \mapsto \text{"}abc\text{"}\}$ to indicate that σ assigns the term $f[\text{"}42\text{"}]$ to the variable named X, "abc" to Y, and no value to all other variables.[9]

For later use, we define the *restriction* $\Sigma\mid_U$ of a substitution set Σ to a set of variables U. Intuitively, the substitutions in $\Sigma\mid_U$ are for the variables of U the same as in Σ, and undefined ($\perp$) for all other variables. Formally,

$$\Sigma\mid_U = \{\sigma' \mid \exists \sigma \in \Sigma\ \forall x.\ \sigma'(x) = \sigma(x) \text{ if } x \in U,\ \sigma'(x) = \perp \text{ otherwise }\}$$

Event sequences. Event sequences $s = \langle a_1, \ldots a_n \rangle_b^e$ are sequences of *temporally ordered* atomic events $a_i = d_i^{r_i}$ together with a beginning time b and an ending time e of the sequence. It is required that beginning time b is earlier than or equal to all reception times in the sequence, and ending time e is later than or equal to all reception times in the sequence. Formally: $b \leq rcp(a_1) < rcp(a_2) < \cdots < rcp(a_n) \leq e$

Beginning and ending time of an event sequence which is (part of) an answer to a composite event query correspond to the notion that a composite event stretches over a time interval — starting with the beginning time of the event sequence. The composite event occurs, i.e., the rule attached to the query fires, at the ending time of the event sequence.

Note that the stream of incoming atomic events (*event stream* for short) introduced above is an event sequence. It contains *all* atomic events that happened in its duration interval, a thing that cannot be said for arbitrary event sequences.

Also note that the empty event sequence over some time interval $[b..e]$ —written $\langle\rangle_b^e$— is considered a legal event sequence by the definition above.

In simple cases, such as an `and`-conjunction between atomic events, the beginning time b will be the reception time r_1 of the first atomic event in the sequence, and the ending time e the reception time r_n of the last atomic event in the sequence. This is however *not* true in general.

Again, to avoid super- and subscript notation, define $begin(s) = b$ and $end(s) = e$ for the beginning and ending times of an event sequence $s = \langle a_1, \ldots a_n \rangle_b^e$.

Subsequences. To be used in an answer, an event sequence s must be a subsequence of the event stream $\mathcal{E}$, that is, it contains only atomic events from the event stream. To formalize this requirement we introduce a subsequence relation between event sequences, represented with the (round) inclusion sign $\subseteq$. The requirement is then $s \subseteq \mathcal{E}$. However, later in this section the right hand side can also be an arbitrary event sequence s' (other then $\mathcal{E}$), giving $s \subseteq s'$. Formally we define:

$\langle a_1, \ldots a_n \rangle_b^e \subseteq \langle a'_1, \ldots a'_m \rangle_{b'}^{e'}$ if and only if

- $\{a_1, \ldots a_n\} \subseteq \{a'_1, \ldots a'_m\}$, and
- $b' \leq b$ and $e \leq e'$.

[9] This is a simplified definition. In [64], substitutions map *all* variables to *construct* terms (where $X \mapsto X$ corresponds to no assignment in our simplified definition). There, the slightly more complicated definition is necessary to define the semantics of Xcerpt rules which include a construction part (cf. [64]). The definition of the semantics of XChange event queries is the same, not matter which definition of substitution is used. Note however that a definition of semantics of whole XChange rules needs to accommodate construction and thus should use the definition from [64].

An event sequence can also be a complete subsequence of the event stream, that is, it contains *all* atomic events from the event stream that lie between the sequence's beginning and ending time. This requirement is written with a squared inclusion sign $s \sqsubset \mathcal{E}$. Again, it is convenient to define the complete subsequence relation between arbitrary event sequences ($s \sqsubset s'$):

$\langle a_1, \dots a_n \rangle_b^e \sqsubset \langle a'_1, \dots a'_m \rangle_{b'}^{e'}$ if and only if

- $\{a_1, \dots a_n\} = \{a'_i \mid b \leq rcp(a'_i) \leq e,\ 1 \leq i \leq m\}$, and
- $b' \leq b$ and $e \leq e'$.

Note that $s \sqsubset \mathcal{E}$ implies $s \subseteq \mathcal{E}$. Obviously, both relations $\subseteq$ and $\sqsubset$ are reflexive and transitive.

Union of event sequences. For later use when defining the answer relation $\triangleleft$, we also define the *union* of two event sequences. The result $s'' = s \cup s'$ is the event sequence containing all events from s and s'. The resulting event sequence stretches over a time interval covering the intervals of s and s'. Formally:

$\langle a_1, \dots a_n \rangle_b^e \cup \langle a'_1, \dots a'_m \rangle_{b'}^{e'} =_{def} \langle a''_1, \dots a''_p \rangle_{b''}^{e''}$ where

- $\{a''_1, \dots a''_p\} = \{a_1, \dots a_n\} \cup \{a'_1, \dots a'_m\}$, and
- $b'' \leq a''_1 < \dots < a''_p \leq e''$, and
- $b'' = \min\{b, b'\}$ and $e'' = \max\{e, e'\}$

The operation $\cup$ is obviously associative. As usual, $\bigcup_{1 \leq i \leq n} s_i$ is shorthand for $s_1 \cup \dots \cup s_n$.

2.2.4.3 Relating Queries and Answers

We will now define a relation expressing when a query q is successfully answered by some answer (s, Σ). The relation depends on the stream $\mathcal{E}$ of incoming atomic events. We write $q \triangleleft_{\mathcal{E}} s, \Sigma$ to express that query q is *answered by* answer (s, Σ) under the event stream $\mathcal{E}$. Recall that we allowed dropping the parentheses so that $q \triangleleft_{\mathcal{E}} s, \Sigma$ is just short form of $q \triangleleft_{\mathcal{E}} (s, \Sigma)$.

It deserves some justification why we use this special answering relation $\triangleleft_{\mathcal{E}}$ instead of some "normal" model theoretic satisfaction relation, i.e., something along the lines of $\mathcal{M} \models q[\sigma]$ (or rather $\mathcal{E} \models q[\Sigma]$). First of all, in XChange an answer is not given by just applying a substitution (set) to a query q; the event sequence part s of an answer needs to be accommodated. Secondly, this event sequence can, due to partial matches (e.g., `andthen [[...]]`), contain more atomic events than actually specified through an event query's constituent atomic event queries. Finally, negation is different from classical negation: `without` q tells us that *for all* Σ the query q cannot be answered; classical negation "$\neg q$" would only tell us that there is *some* Σ such that $\mathcal{E} \not\models q[\Sigma]$.

For a clean definition of answering substitution sets, we need the notion of negative and positive polarity of variable occurrences in a composite event query. Intuitively, a variable occurrence in a composite event query is called *negative* if evaluation of the composite event query will yield a value for this variable occurrence. We also say that the variables occurs in a *defining* position. Otherwise the variable occurrence is said to be *positive* (or *non-defining*). For example, in

```
without {
    c [ var X, var Y ]
} during {
    andthen [ a{var X}, b{var X} ]
}
```

the only occurrence of Y is positive (non-defining), and also the first occurrence of X: both are inside a `without` and thus a successful evaluation of the query will not yield values for them. The other two occurrences of X are negative (defining): a successful evaluation of the query will yield values for them.
Formally, an occurrence of a variable is positive (non-defining) if

- it occurs inside the left query of an XChange `without` (i.e., in q_1 of `without` q_1 `during` q_2 or `without` q_1 `during` [b..e]), or if

- it is a positive occurrence according to [64] in the Xcerpt query term of an atomic event query (e.g., it occurs in Xcerpt `without`).

It is negative (defining) otherwise.

Assuming a standardization of variables in queries, let V be the set of all variables having at least one negative occurrence. The variables of V are those that will be assigned values, and can thus be used in the condition- and action-part of XChange rules.

The answering relation $\lhd_{\mathcal{E}}$ is defined inductively on the query q. The induction base is an atomic query, the induction step uses case distinction on the top-level query operator or temporal restriction.

For triggering an XChange rule attached to an event query q only those answers (s, Σ) with *maximal* substitution sets Σ are considered. Note that there can be several maximal substitution sets for the same event sequence s.

Intuitively, for an answer (s, Σ) to an event query q, a substitution set Σ is said to be *maximal* (w.r.t. a query q and the event sequence s) if there is no substitution set Φ such that (s, Φ) answers q and Σ is a proper subset of Φ.

Formally, a substitution set Σ is maximal w.r.t. some property $P(\Sigma)$ and a set of variables U if and only if for all substitution sets Φ with $P(\Phi)$ we have $\Phi \mid_U \subseteq \Sigma \mid_U$ (see also [64]). For our purpose we will have $P(\Sigma) \equiv q \lhd_{\mathcal{E}} s, \Sigma$ for fixed q, $\mathcal{E}$, s and $U = V$ for V the set of all variables with at least one negative (defining) occurrence in q.

Atomic Query. $q \in \mathcal{T}^q$, i.e., q is an atomic event query ($\mathcal{T}^q$ denotes the set of all possible Xcerpt query terms). Then, $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- for all free variables X occurring in q and all $\sigma \in \Sigma$, $\sigma(X)$ is defined and respects all variable restrictions on X in q (expressions of the form `var X -> q'` inside q, see [64, chapter 7]).

- $\forall t \in \Sigma(q) : \quad t \preceq d$

- $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}$.

The first and the second line of the definition can be summarized as follows: the query q matches the data term d under all substitutions from the substitution set Σ.

The second line deserves some explanation. For a substitution σ and a query q, $\sigma(q)$ denotes the *ground query term*[10] resulting from replacing all variable occurrences in q by their value according to σ (Note that for every variable X in q, the value $\sigma(X)$ is defined due to line 1). For a substitution set Σ and a query q, $\Sigma(q)$ denotes the set of all ground query terms resulting from the substitutions in Σ, i.e., $\Sigma(q) = \{\sigma(q) \mid \sigma \in \Sigma\}$. The relation $\preceq$ between a ground query term t and a data term d denotes that t *simulates* in d; that is to say, t and d "match". The necessary formal definitions can, again, be found in [64].

Note that we use a sequence containing a single atomic event here as the answer to the atomic event query. By this, we make the atomic event query simply a special case of a composite event query, instead of giving it extra treatment; this makes especially sense since we want to use it as the base of our inductive definition.

Conjunction. $q = \texttt{and}\{q_1, \ldots\ q_n\}$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist event sequences $s_1, \ldots s_n$ such that

- $q_i \lhd_{\mathcal{E}} s_i, \Sigma$ for all $1 \leq i \leq n$
- $s = \bigcup_{1 \leq i \leq n} s_i$

Note that from s being the union of the s_i it immediately follows that $begin(s) = \min_i begin(s_i)$ and $end(s) = \min_i end(s_i)$. This means that a composite event built with `and` stretches over the time interval covering all constituent events.

Instead of defining the `and`-operator with variable arity, it would also suffice to define it binary and introduce the following rewriting rules to reduce variable arity to binary: $\texttt{and}\{q_1\} \mapsto q_1$, $\texttt{and}\{q_1, q_2, q_3 \ldots q_n\} \mapsto \texttt{and}\{q_1, \texttt{and}\{q_2, q_3, \ldots q_n\}\}$. We will make use of this possibility when defining more complicated variable arity operators such as `andthen`.

Disjunction. $q = \texttt{or}\{q_1, \ldots\ q_n\}$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q_i \lhd_{\mathcal{E}} s, \Sigma$ for some $1 \leq i \leq n$

This is the simplest composition operator. The `or`-query simply "inherits" its answer(s) from the constituent queries.

Sequence. Defining `andthen` with variable arity directly would require lots of confusing notation. Instead, we will first define the binary cases for `andthen` with both complete `[ ]` and incomplete `[[ ]]` specifications, and give rules for reducing the variable arity operator to the binary case afterwards.

Case $q = \texttt{andthen}[q_1, q_2]$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist event sequences s_1 and s_2 such that

- $q_i \lhd_{\mathcal{E}} s_i, \Sigma$ for $i = 1, 2$
- $s = s_1 \cup s_2$
- $end(s_1) < begin(s_2)$

The requirements imply that $begin(s) = begin(s_1)$ and $end(s) = end(s_2)$.

[10] A term is called *ground* if it does not contain variables.

Case $q = \texttt{andthen}[[q_1, q_2]]$.
$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist event sequences s_1, s', and s_2 such that

- $q_i \lhd_{\mathcal{E}} s_i, \Sigma$ for $i = 1, 2$
- $s = s_1 \cup s' \cup s_2$
- $end(s_1) \leq begin(s_2)$
- $begin(s') = end(s_1)$ and $end(s') = begin(s_2)$
- $s' \sqsubset \mathcal{E}$

Again, the requirements imply that $begin(s) = begin(s_1)$ and $end(s) = end(s_2)$.

The event sequence s' serves to capture the events happening between the answers s_1 and s_2 to q_1 and q_2, respectively — as demanded by the partial match `[[ ]]`. The last line of the definition requires that *all* those events from the event stream $\mathcal{E}$ are contained and no event is left out (recall that with $\sqsubset$ we denote a *complete* subsequence of the event stream).

Case $q = \texttt{andthen}[q_1, q_2, q_3, \ldots q_n],\ n > 2$.
Apply the rewriting rule $\texttt{andthen}[q_1, q_2, q_3, \ldots q_n] \mapsto \texttt{andthen}[q_1, \texttt{andthen}[q_2, q_3, \ldots q_n]]$.

Case $q = \texttt{andthen}[[q_1, q_2, q_3, \ldots q_n]],\ n > 2$.
Apply the rewriting rule $\texttt{andthen}[[q_1, q_2, q_3, \ldots q_n]] \mapsto \texttt{andthen}[[q_1, \texttt{andthen}[[q_2, q_3, \ldots q_n]]]]$.

Absolute Temporal Restriction.

Case $q = q'\ \texttt{in}\ [b..e]$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q' \lhd_{\mathcal{E}} s, \Sigma$
- $b \leq begin(s)$ and $end(s) \leq e$

Case $q = q'\ \texttt{before}\ e$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q' \lhd_{\mathcal{E}} s, \Sigma$
- $begin(\mathcal{E}) \leq begin(s)$ and $end(s) \leq e$

Note that the time points b and e are treated as already interpreted objects, not their actual syntactical representation. Albeit being formally a little unsound, this enhances the readability, as we dispense with an interpretation function for time objects. To be cleaner, one could take b and e to be syntactic objects in "$q = q'\ \texttt{in}\ [b..e]$", and replace b and e with $\mathcal{I}(b)$ and $\mathcal{I}(e)$ in the body of the definition, where $\mathcal{I}$ is an interpretation function for time objects.

Relative Temporal Restriction. $q = q'\ \texttt{within}\ w$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q' \lhd_{\mathcal{E}} s, \Sigma$
- $end(s) - begin(s) \leq w$

As in the previous case, the time length w is an already interpreted object, not the actual syntactical representation.

Variable Restriction. $q =$ `var` $X \rightarrow q'$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q' \lhd_{\mathcal{E}} s, \Sigma$
- for all $\sigma \in \Sigma$: $\sigma(X) =$ `xchange:event-seq[` $d_b, d_1, \ldots d_n, d_e$ `]` where $s = \langle d_1^{r_1}, \ldots d_n^{r_n} \rangle_e^b$, $d_b =$ `xchange:beginning-time[`b`]`, $d_e =$ `xchange:ending-time[`e`]`.

Exclusions.

Case $q =$ `without` $\{q_1\}$ `during` $\{q_2\}$. $q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q_2 \lhd_{\mathcal{E}} s, \Sigma$
- For all $s' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s')$ and $end(s') \leq end(s)$ and all Σ' with $q_1 \lhd_{\mathcal{E}} s', \Sigma'$ it holds that $\Sigma\mid_V \cap \Sigma'\mid_V = \emptyset$

Remember that V is the set of all variables having at least one negative (defining) occurrence, i.e., occur at least once outside of a `without` operator. Variables occurring only positively (i.e., variables that are not members of V) are implicitly universally quantified: for all possible values of these variables, the query q_1 *must not* be successful. The last line of the definition captures this.

Case $q =$ `without` $\{q_1\}$ `during` $[b..e]$. Variation on the case above: remove $q_2 \lhd_{\mathcal{E}} s, \Sigma$ and replace $begin(s')$ with b and $end(s')$ with e.

Quantifications.

Case $q =$ `times` n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $\{q''\}$.
(Note that the keyword `any` is dropped in the case $k = 0$ and that $n \geq 1$.)
$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist n event sequences $s_1, \ldots s_n$ and substitution sets $\Sigma_1, \ldots \Sigma_n$, and an event sequence s'' such that

- $s = s'' \cup \bigcup_{1 \leq i \leq n} s_i$
- $q'' \lhd_{\mathcal{E}} s'', \Sigma$
- $q' \lhd_{\mathcal{E}} s_i, \Sigma_i$ for all $1 \leq i \leq n$
- $begin(s'') \leq begin(s_i)$ and $end(s_i) \leq end(s'')$ for all $1 \leq i \leq n$
- $\Sigma_i \mid_{V \setminus \{X_1, \ldots X_k\}} = \Sigma_j \mid_{V \setminus \{X_1, \ldots X_k\}}$ for all $1 \leq i < j \leq n$
- $\Sigma \subseteq \bigcup_{1 \leq n} \Sigma_i$
- Σ_i is maximal (w.r.t. V and $q_1 \lhd_{\mathcal{E}} s_i, \Sigma_i$) for all $1 \leq i \leq n$
- $s_i \neq s_j$ for all $1 \leq i < j \leq n$
- if there exists an $s' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s')$ and $end(s') \leq end(s)$, and a Σ' with $\Sigma' \mid_{V \setminus \{X_1, \ldots X_k\}} = \Sigma \mid_{V \setminus \{X_1, \ldots X_k\}}$ such that $q' \lhd_{\mathcal{E}} s', \Sigma'$, then $s' = s_i$ and $\Sigma' \mid_V \subseteq \Sigma_i \mid_V$ for some $1 \leq i \leq n$

For q to be successfully answered, q' must be answered by at least n *different* answers (s_i, Σ_i) (lines 3, 7, and 8). The substitution sets Σ_i must agree on all variables, except the existentially quantified variables $X_1, \ldots X_k$ (line 5). The last line requires that there are no more than the n answers.

Case $q = \texttt{times atleast } n \texttt{ any var } X_1, \ldots, \texttt{var } X_k \ \{q'\} \texttt{ during } \{q''\}$.
$q \triangleleft_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $p \geq n$ event sequences $s_1, \ldots s_p$ and substitution sets $\Sigma_1, \ldots \Sigma_p$, and an event sequence s'' such that

- Conditions from above with n replaced by p hold.

Case $q = \texttt{times atmost } n \texttt{ any var } X_1, \ldots, \texttt{var } X_k \ \{q'\} \texttt{ during } \{q''\}$.
$q \triangleleft_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $1 \leq p \leq n$ event sequences $s_1, \ldots s_p$ and substitution sets $\Sigma_1, \ldots \Sigma_p$, and an event sequence s'' such that

- Conditions from above with n replaced by p hold.

Cases $q = \texttt{ times (atleast|atmost)? } n \texttt{ any var } X_1, \ldots, \texttt{var } X_k \ \{q'\} \texttt{ during } [b..e]$.
Variations on the above cases: replace $q'' \triangleleft_{\mathcal{E}} s'', \Sigma$ with $s'' = \langle\rangle_b^e$. (This is a little "trick" where we use that empty sequences still have a duration. We could have defined the `without` $\{q_1\}$ `during` $[b..e]$-case in the same manner, but didn't in order to illustrate the effects it has.)

Multiple Inclusions and Exclusions.

Case $q = m \texttt{ of any var } X_1, \ldots \texttt{var } X_k \ \{q_1, \ldots q_n\} \texttt{ during } \{q''\}$.
(Note that the keyword `any` is dropped in the case $k = 0$ and that $m \geq 1$.)
$q \triangleleft_{\mathcal{E}} s, \Sigma$ holds if and only if there exist m event sequences $s_1, \ldots s_m$ and substitution sets $\Sigma_1, \ldots \Sigma_m$ and a injective mapping $\iota : \{1, \ldots m\} \to \{1, \ldots n\}$, and an event sequence s'' such that

- $s = s'' \cup \bigcup_{1 \leq i \leq m} s_i$
- $q'' \triangleleft_{\mathcal{E}} s'', \Sigma$
- $q_{\iota(i)} \triangleleft_{\mathcal{E}} s_i, \Sigma_i$ for all $1 \leq i \leq m$
- $begin(s) \leq begin(s_i)$ and $end(s_i) \leq end(s)$ for all $1 \leq i \leq m$
- $\Sigma_i \mid_{V \setminus \{X_1, \ldots X_k\}} = \Sigma_j \mid_{V \setminus \{X_1, \ldots X_k\}}$ for all $1 \leq i < j \leq m$
- $\Sigma \subseteq \bigcup_{1 \leq m} \Sigma_i$
- Σ_i is maximal (w.r.t. V and $q_1 \triangleleft_{\mathcal{E}} s_i, \Sigma_i$) for all $1 \leq i \leq n$
- if there exists an $s' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s')$ and $end(s') \leq end(s)$, and a Σ' with $\Sigma' \mid_{V \setminus \{X_1, \ldots X_k\}} = \Sigma \mid_{V \setminus \{X_1, \ldots X_k\}}$ and a $1 \leq j \leq n$ such that $q_j \triangleleft_{\mathcal{E}} s', \Sigma'$, then $j = \iota(i)$ for some $1 \leq i \leq m$

For q to be successfully answered, at exactly m out of the n queries $q_1, \ldots q_n$ have to be answered; the injection ι tells which. The third line of the definition demands that there are at least m answered queries, the last line that there are no more than m. Existential quantification of variables with `any` is taken care for in line 5.

Case $q =$ `atleast` m `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $\{q''\}$.
$q \triangleleft_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $p \geq m$ event sequences $s_1, \ldots s_p$ and substitution sets $\Sigma_1, \ldots \Sigma_p$ and an injective mapping $\iota : \{1, \ldots p\} \to \{1, \ldots n\}$, and an event sequence s'' such that

- Conditions from above with n replaced by p hold.

Case $q =$ `atmost` m `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $\{q''\}$.
$q \triangleleft_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $1 \leq p \leq m$ event sequences $s_1, \ldots s_p$ and substitution sets $\Sigma_1, \ldots \Sigma_p$ and a injective mapping $\iota : \{1, \ldots p\} \to \{1, \ldots n\}$, and an event sequence s'' such that

- Conditions from above with n replaced by p hold.

Cases $q =$ (`atleast`|`atmost`)? m `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $[b..e]$.
Variations on the above cases: replace $q'' \triangleleft_{\mathcal{E}} s'', \Sigma$ with $s'' = \langle\rangle_b^e$.

2.2.5 Operational Semantics: Incremental Evaluation

Having described XChange's event queries both informally and formally, we now turn to look at their evaluation. For this, we start by outlining requirements and considerations for the evaluation of event queries (Section 2.2.5.1). Evaluation of atomic event queries is conceptually pretty straightforward and we touch this only briefly (Section 2.2.5.2) before turning to the heart of this section, the evaluation of composite event queries (Section 2.2.5.3). Ideas on optimizations of the event query evaluation conclude this part of the report (Section 2.2.5.4).

2.2.5.1 Requirements and Considerations

The setting for event query evaluation is as follows: We have a number of (atomic and composite) event queries $q_1, \ldots q_n$ currently registered in the system. Every time a new atomic event a is received, we need to check for every event query q_i if it can be answered and should trigger the execution of its rule. The evaluation of the query q_i has to consider the new atomic event a and — if q_i is composite — also some atomic events received previously, i.e., some part of the stream of incoming events the query has "seen" so far.

Incremental Evaluation Preferably, evaluation of event queries should be performed in an incremental manner. That is, we want to save work done in the evaluation of an event query on some incoming atomic event for future evaluation on future incoming events. To give an example, suppose evaluating the composite event query

$q =$ `and{ a{var X}, b{var Y} } within 1h`

on the stream

```
< a{1}, b{2} >
```

of incoming events.

When `a{1}` is received, we need to evaluate q for the first time. In doing so, we check whether `a{1}` answers any of its constituent atomic event queries `a{var X}`, `b{var Y}`; indeed

it answers the first, but not the second. Since a{1} is the first atomic event we receive, we do not need to check if the second had an answer previously and know that the composite and-query q is not answered, yet.

When the next b{2} is received, we need to evaluate q again. In doing so, we obviously have to check whether b{2} answers a{var X} or b{var Y}; indeed it answers the second, but not the first. Now, the composite query is successful if the first constituent query has been answered by a previous incoming atomic event. This is the case here: a{1} has answered a{var X}.

The last fact has already been established in a previous evaluation of q. In an incremental evaluation of q we "remember" this fact and use it in the current evaluation. In contrast, a non-incremental evaluation would have to perform the check whether the previously received a{1} answers any of a{var X}, b{var Y} all-over again.

Constant Evaluation Cost per Incoming Atomic Event. The cost of an evaluation of a (legal) event query on some incoming atomic event should be kept roughly constant. That is, it should not grow with the number of atomic events received so far. This guarantees good performance and predictable response times for an XChange-aware Web site.

Two things are the key to fulfilling this requirement. First, the incremental evaluation discussed above. It memorizes work done in previous evaluations in order to not redo it. Second, we need to avoid that the size or number of composite events caused by a single incoming event grows (w.r.t. to the number of previously received atomic events). The bounded event life-span for legal event queries gives us a practical bound on the size and number of composite events since we consider only atomic events inside this temporal bound in forming the composite event.[11]

To illustrate the latter, consider as a counter-example the non-legal event query

```
and { a{var X}, b{var Y} }
```

being evaluated on

```
< a{1}, b{1}, a{2}, b{2}, a{3}, b{3}, ... >
```

Upon reception of b{1}, we have only one composite event, event-seq [a{1}, b{1}]. Upon reception of b{2}, we have already two composite events, event-seq [a{1}, b{2}] and event-seq [a{2}, b{2}]. Upon reception of b{3}, we have three composite events, and so on. The number of composite events grows linearly in the number of previously received atomic events. Inherently, the processing of each single incoming b{i} thus becomes slower and slower over time.

Utilization of Bounded Event Life-Span. Due to the definition of legal event queries, they can be evaluated with events with a bounded life-span. The actual evaluation must make use of this language feature and discard events whose life-span has expired.

[11] Note that this is only a *practical* bound for the number and size of composite events. Theoretically we could have atomic events a_n arrive at times $t_n = 1 - \frac{1}{n}$ and thus squeeze infinitely many events in the time-interval [0..1]. Or we could have atomic events a_n arriving at regular intervals (i.e., $t_n = n$) but with a growing size $s_n = n$. Practically however, factors like network bandwidth and processor speed limit the data —and thus the number and size of atomic events— received in some time interval.

Treatment of Variable Assignments. As we have seen in the introduction of the event query language (Section 2.2.3) and the declarative semantics (Section 2.2.4), answers to event queries (and also answers to the constituent event queries of a composite event query) include each a substitution set assigning values to the free variables in a query. The query evaluation has the task of finding the *maximal* substitution set.

This requirement is, of course, somewhat obvious. Still, it does deserve explicit mentioning: Related work on the evaluation of composite event queries has focused on the evaluation of the composition operators and possibly also the evaluation of temporal constraints. Prior to work on XChange, treatment of logical variables (i.e., variables that enforce equality of their assigned values if they occur in different places of an atomic or composite event query) has, to the best of our knowledge, not been given much thought. Existing algorithms for composite event query evaluation cannot be simply reused. They need at least significant adaption. The handling of logical variables is described later in this report, in the context of the general framework.

2.2.5.2 Evaluation of Atomic Event Queries

Atomic event queries appear as event queries by themselves or as leaves in the query trees of composite event queries. Evaluation of a given atomic event query is easy enough: for each incoming event message we test whether it matches with the atomic event query using Simulation Unification [23]. We need not to know any of the inner workings of Simulation Unification; it suffices that it is either successful or unsuccessful. In the successful case it delivers a set of substitutions for the free variables in the atomic event query. Together with the event message, this forms an answer to the atomic event query.

Whenever an event message comes in, it has to be evaluated against every atomic event query currently registered as stand-alone or as constituting part of a composite event query. As there are typically quite many such atomic event queries, efficient evaluation can be crucial for the system performance and should be the starting-point of any serious discussion on optimization. We return to this later in Section 2.2.5.4.

2.2.5.3 Evaluation of Composite Event Queries

Evaluation of composite event queries, sometimes also called composite event detection, is best done in an incremental manner by maintaining a partial evaluation that is update whenever a new event message comes in.

Related Work on Composite Event Detection. Extensive research on the issue of (incremental) composite event detection has been conducted in the area of active database systems. The following three attempts to composite event detection are popular. They can be distinguished by how they represent the partial event query evaluation, or in other words the "progress" in the detection of a composite event [28, Chapter 5.3]:

- **Finite State Automata.** A partial event query evaluation can be represented as a finite state automaton. The states signify the "progress" made in the detection of a composite event. State transitions are triggered by incoming atomic events. If a final state is reached, the composite event has been detected. Finite State Automata are popular since many event query languages bear a strong resemblance to regular expressions and construction of an automaton from a given regular expression is a well-understood

problem. An example here is COMPOSE, which has been developed with the Ode (active) object-oriented database [38, 37, 39].

It is conceivable to use other, less restricted forms of automata, too, say push down automata. However, to our knowledge, no such attempt has been made.

- **Petri Nets.** In similar fashion, a partial event query evaluation can be represented as a (special type of) Petri net. In the active object-oriented database system SAMOS, so-called SAMOS Petri Nets (S-PN), which are based on Colored Petri Nets, are used [35, 36]. The main components of an S-PN are places (input places, output places, and auxiliary places), transitions, and arcs (wiring transitions and places).

  For some given composition operator, tokens on input places model successful evaluation of a constituting event query, tokens on output places successful evaluation of the composite query. The auxiliary places are used, together with the transitions and the wiring provided by the arcs, to model dependencies between event occurrences (e.g., "$E1$" has to occur *before* "$E2$").

- **Query Trees with Bottom-Up Flow of Events.** A different approach uses a query tree (or graph) that mirrors the structure of the syntax tree (operator tree) of a given composite event query. The leaf nodes represent atomic event queries, the inner nodes represent composition operators. Atomic events are injected at the leaf nodes, and we have a bottom-up flow of (partially composed) events in the tree.

  For a given composition operator, its inner node has several children, one for each constituting event query. The children provide input data: the successful evaluations of the constituting event queries. Additionally, each inner node has some kind of storage facility, used to memorize events (or other information) possibly needed in future evaluations.[12] For example, in an evaluation of `andthen[a, b]` on `a`, the event `a` does not make a composite event occur, but has to be memorized for the future (when a `b` arrives). When the inner node detects a composite event from the input events and the memorized events, it outputs the composite event as input to its parent node.

  The tree-based approach seems to be the most widely adopted; it has been presented first in SNOOP [25] and been used in a number of systems subsequently, e.g., GEM [47], EPS [53], and ruleCore (see Appendix A and [62]).

  The basic idea for this kind of incremental evaluation can be found in the *rete algorithm* [34]. It describes an incremental forward-chaining evaluation for use in inference systems where over time new facts are told to the system.

Composite Event Detection in XChange. Composite event detection in XChange uses a tree-based approach. While approaches based on automata and Petri nets come with computation models that are well understood in practice and theory, one has to devise new, specialized algorithms for the data-flow in the tree-based approach. Still, the tree-based approach scores over the others in the following points:

[12] Depending on the event language and its notion of answers, there might be cases where one actually does not have to memorize events. For some sorts of sequence operators, for example, it can suffice to simply "activate" the right child when an event from the left child has been detected. This approach is described in [28, chapter 5.3]. In the tree-based approach used in [25], for example, nodes do store events.

- **Reflection of Query Structure.** The query evaluation tree reflects the structure of the query (more precisely, the query's operator tree). It is thus easy to comprehend and makes an implementation easier to debug.

  In contrast, an automaton constructed from a query bears little resemblance to the query structure. The construction is potentially error-prone; the same holds for Petri nets.

- **Efficiency and Feasibility of Implementation.** The tree-based approach has been shown to be reasonably efficient, and lends itself easily to optimizations such as query rewriting or exploiting similarities between different queries. As described in [30, Chapter 8], implementation is not too difficult once the needed data structures and algorithms are understood.

  In contrast, implementation of Petri nets can be complex and quite inefficient as argued in [53]. Implementation of (Finite State) Automata is well-understood and should usually be computationally efficient; however, there is a danger that the number of states is exponential in the size of the query, leading to an expensive construction and bad memory requirements. (Lazy automaton construction can help here but usually makes implementation significantly harder.)

- **Versatility.** It has been shown that the tree-based approach can be adapted to various requirements. GEM [47] presents a version that can deal with delays in event reception due to unsynchronized clocks. EPS [53] presents a version that also deletes events when they time out. In this work we present a version that also supports deletion of events and, moreover, deals with variable assignments (cf. 2.2.5.1).

  To our knowledge there has been little work on extending Finite State Automata or Petri nets to accommodate requirements such as delayed event reception or variables. (The Petri nets-based SAMOS [36] provides limited access to data contained in events, but does not provide an easy way to accommodate substitutions or substitution sets arising from atomic event queries.)

Tree-Based Representation of Partial Event Query Evaluations. Parsing and compiling an event query results in an operator tree. The inner nodes of the operator tree implement language constructs such as `and`, `or`, `where`, `within`, `without ... during ...`, `times ... during ...`; the leaves implement atomic event queries. For a given query, we obtain the operator tree in the obvious way.[13] For example the query in Figure 2.1(a) corresponds to the operator tree in Figure 2.1(b).

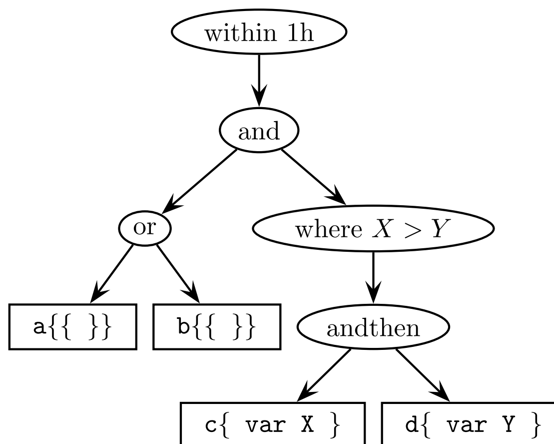
Note that the construct `without` $\{q_1\}$ `during` $\{q_2\}$ is implemented by *one* node with q_1 as left child and q_2 as right child. The same applies to `times` *mult* $\{q_1\}$ `during` $\{q_2\}$ and *mult* `of` $\{q_1\}$ `during` $\{q_2\}$. In the following we will only consider the operators `and`, `or`, and `andthen` in their binary forms; variable arity forms can be reduced to binary ones as discussed in Section 2.2.4.3.

In order to use the operator tree in an incremental evaluation, some inner nodes are extended for storing composite events (that is, tuples consisting of an event sequence and a substitution

[13] In general, e.g., in relational databases, it is not so obvious how to obtain an operator tree from a query as there is not necessarily a one-to-one correspondence between language constructs and physical operators. Also, there are usually many operator trees for a query and these can differ significantly in efficiency. Here however, we have a one-to-one correspondence of language constructs and operators, so we can easily obtain a basic operator tree that resembles the query's syntactic structure closely.

```
and {
    or {
        a {{ }},
        b {{ }}
    },
    andthen [
        c { var X },
        d { var Y }
    ] where { var X > var Y }
} within 1h
```

(a) A (composite) event query



(b) The query's operator tree

Figure 2.1: A (composite) event query and its operator tree

set). The node stores all composite events it received from its children that might be needed in the future to compose the answer event. The details depend on the operator and should be chosen so as to allow an efficient event composition.

Let us consider the storage for `or`, `andthen`, and `and`.

An operator node for `or` does not need any storage at all; when any of `or`'s two children detects an event, this event is immediately handed over to the `or`'s parent node.

An operator node for `andthen` must only store events from its left child, e.g., as a list of events sorted by their ending time. When an event from its right child arrives, this event can immediately be combined with all the stored events from the left child to form the composite events that are handed over to the `andthen`'s parent.

An operator node for `and` has to store events both from its left child and right child. If an event from the left child arrives, it has to be combined with all the stored events previously received by the right child, and vice versa. Accordingly, a reasonable way for the event storage in an `and`-node is to maintain one list of events from the left child and another list of events from the right child.

Other operator nodes might be better implemented with more advanced storage structures than simple lists. For example, `times` *mult* $\{q_1\}$ `during` $\{q_2\}$ could use for its right child (q_2) a simple list, but for its left child (q_1) a table organizing events by their substitution sets (remember the definition of `times`: the substitution sets for events matching q_1 have to be equal to provide an answer to the `times`-query).

Note that it usually suffices to store *references* to the events, not copies. Especially, for the potentially large XML documents of event messages we only need to maintain a single copy in memory, and we can then use references in the operator nodes. The event messages can even be stored in secondary storage, e.g., on a hard disk: for the event query evaluation we will not have to access the event messages themselves, only their substitution sets.

So far in this discussion, we have said that the composed composite events are to be handed over to the parent. The root of the operator tree has, however, no parent. Composite events handed over by the root to its "non-existent parent" are those composite events that trigger

execution of the event query's rule. Instead of this non-existing parent, we can also imagine a virtual root node that receives the events from the real root and takes care of triggering rule execution.

Bottom-Up Data-Flow for Event Detection. In the operator tree representing some event query, incoming event messages are injected at the leaf nodes (which correspond to atomic event queries). From there, data in the form of composite events, that is, tuples of an event sequence and a substitution set, flow upwards in the tree. Composite events leaving the root node trigger execution of the rule attached to the evaluated event query.

We will now illustrate this bottom-up data-flow in the operator tree. The (binary) and-operator node provides with a simple but sufficiently interesting example.

When the and-operator node is being evaluated we have the following information available:

- events detected in the current evaluation by the left child (newL),
- events detected in the current evaluation by the right child (newR), and
- stored events from previous evaluations, one list for the left child (oldL) and one list for the right child (oldR).

Here and in the following, "event" will always mean a composite event, that is, a tuple (s, Σ) of an event sequence and a substitution set. As in Section 2.2.4, an atomic event leaving the leaf nodes (representing atomic event queries) in the tree can be treated as a special case of a composite event where the event sequence contains only one event message.
The and-node now has to do two things:

- compute all events that can be composed in the current evaluation from the given four sets of events newL, newR, oldL, oldR;
- update the event storage for future evaluations, that is, compute event sets oldL', oldR' from newL, newR, oldL, oldR.

The second task is trivial: all available events, the new and the old, can play a role in future evaluations of and. So we have $\mathsf{oldL}' := \mathsf{oldL} \cup \mathsf{newL}$ and $\mathsf{oldR}' := \mathsf{oldR} \cup \mathsf{newR}$. Note that deletion of events due to the bounded life-span is dealt with separately (see next section).

The first task is the interesting one. For every event generated by the right child (either in the current evaluation or in a previous one) we need to check whether there is an event generated by the left child such that the two can agree on a common substitution set. In other words, every tuple $((s_L, \Sigma_L), (s_R, \Sigma_R))$ of events in $\mathsf{newL} \times \mathsf{newR} \cup \mathsf{oldL} \times \mathsf{newR} \cup \mathsf{newL} \times \mathsf{oldR}$ needs to be checked for a suitable common substitution set Σ. If we find one (we will see: Σ must not be empty), the tuple generates the (composite) and-event $(s_L \cup s_R, \Sigma)$. Note that we need not check tuples in $\mathsf{oldL} \times \mathsf{oldR}$; these have already been checked in some previous evaluation.

We are now left with the following task: We are given two substitution sets Σ_L and Σ_R assigning values to all free variables V_L and V_R in their respective queries. (We will assume that the substitutions in Σ_L assign only values to variables from V_L and are undefined for other variables; analogously for Σ_R.) From this we have to compute a maximal "common" substitution set Σ. Σ must assign values to all free variables of both queries $V = V_L \cup V_R$ in a fashion that it does not "contradict" Σ_L or Σ_R. More precisely we must have $\Sigma\ |_{V_L} \subseteq \Sigma_L\ |_{V_L}$

and $\Sigma\mid_{V_R}\subseteq\Sigma_R\mid_{V_R}$ and Σ maximal. Here, the substitution set $\Sigma=\emptyset$ signifies that Σ_L and Σ_R contradict each other and we cannot form a composite event.[14]
Σ can be computed as some special sort of natural join $\Sigma=\Sigma_L\bowtie\Sigma_R$ where

$$\Sigma_1\bowtie\Sigma_2=\{\sigma_1\cup\sigma_2\mid\sigma_1\in\Sigma_1,\sigma_2\in\Sigma_2,\forall X.\ \sigma_1(X)=\sigma_2(X)\vee\sigma_1(X)=\bot\vee\sigma_2(X)=\bot\}.$$

An alternative way to view this it to see substitution sets as logical formulas (constraints on the values a variable can be assigned). Substitutions σ are a conjunction of constraints of the form $X=t$ (X variable, t data term), e.g., $X=\mathtt{f}\{\}\wedge Y=\mathtt{1}$ for $\sigma=\{X\mapsto\mathtt{f}\{\},Y\mapsto\mathtt{1}\}$. Variables not being assigned a defined value do not appear in this conjunction. A substitution set is then a disjunction of its substitutions (it is in disjunctive normal form!). The common substitution set Σ of Σ_L and Σ_R can then be obtained by bringing the conjunction of the formulas for Σ_L and Σ_R into disjunctive normal form: "$\Sigma=\Sigma_L\wedge\Sigma_R$."[15] This view of substitution sets is beneficial if we need to accommodate negations (see Section 2.2.5.3): we can simply also have negated constraints $X\neq t$. Hence, the implementation of event query evaluation in [30] is based on constraints rather than substitution sets.

The join of substitution sets (or, the conjunction of constraints) is the most important operation in the incremental event query evaluation. It is used by almost every composition operator (`or` is an exception) in one form or another. Need for the join is also an aspect that distinguishes event query evaluation in XChange from previous work on composite event detection: previous work usually did not consider variable substitutions obtained from event data at all.

To conclude this discussion on event detection, we will shortly explain the handling of `andthen[ ]`. The `andthen[ ]` operator stores events from its left child (oldL); events from its right child are not needed in future evaluations. Upon evaluation, `andthen[ ]` tests every tuple $((s_L,\Sigma_L),(s_R,\Sigma_R))\in\mathsf{oldL}\times\mathsf{newR}$ for $\mathit{end}(s_L)<\mathit{begin}(s_R)$ and $\Sigma:=\Sigma_L\bowtie\Sigma_R\neq\emptyset$ to generate new composite events $(s_L\cup s_R,\Sigma)$. Note that it is not necessary to check tuples from $\mathsf{newL}\times\mathsf{newR}$ since there we have $\mathit{end}(s_L)=\mathit{end}(s_R)$ and hence automatically $\mathit{end}(s_L)\not<\mathit{begin}(s_R)$.

Top-Down Traversal for Event Deletion. Event detection only adds events to the storages of the nodes in the operator tree. Event deletion has the task of removing those events from the storage that are not needed anymore. Whether an event is needed or not is decided by the bound on the event's life-span provided by the legal event query we are evaluating.

Event deletion depends on the current time and the life-span bound(s) provided by the event query. Time restriction operators (`in`, `before`,`within`) put a bound on all events stored in their subtrees in the operator tree. The same applies to the `during FiniteTimeInterval`-operators.

Accordingly, we traverse the operator tree top-down to delete events. We maintain an absolute time interval $[\mathit{min}...\mathit{max}]$ as a restriction for events. All events that lie in this time interval are still "alive" and can be needed in future event detections. Events that do not (i.e., their begin is earlier than min or their end later than max) are to be deleted.

Initially, when starting traversal at the root node, we place no restriction on events, i.e., $[\mathit{min}..\mathit{max}]:=[-\infty..\mathit{now}]$ (where now is the current time and we have $\mathit{end}(s)\leq\mathit{now}$ for any event (s,Σ) stored in the tree). At each node in our traversal we do the following:

[14] Note the subtle and important difference between $\Sigma=\emptyset$ and $\Sigma=\{\emptyset\}=\{\sigma_\bot\}$ (where $\sigma_\bot(X)=\bot$ for all variables X). The latter is a valid substitution set obtained when no free variables occur in the queries.

[15] Taking up on the previous footnote, $\Sigma=\emptyset$ corresponds to the formula *false*, while $\Sigma=\{\emptyset\}$ corresponds to the formula *true*.

- If the node represents a time restriction operator (`in`, `before`,`within`) or a `during FiniteTimeInterval`-operator, adjust the time restriction $[min..max]$ to $[min'..max']$. The adjustment will be described below. Otherwise leave it unchanged ($[min'..max'] := [min..max]$)
- Test every event (s, Σ) stored in the current node against the time restriction $[min'..max']$; delete it if $[begin(s)..end(s)] \not\subseteq [min'..max']$.
- Traverse all subtrees of the node with the adjusted restriction $[min'..max']$.

Note that, while we start the traversal with no restriction ($[-\infty..now]$), for every legal event query the root node is an operator that immediately adjusts the restriction (definition of legal event queries in Section 2.2.3.4).

We now are left with describing the adjustment of $[min..max]$ to $[min'..max']$ at `in`-, `before`-, `within`-, and `during FiniteTimeInterval`-nodes.

Case Node is `in [`b `..` e`]`.
An event (s, Σ) has to satisfy $[begin(s)..end(s)] \subseteq [min..max]$, the restriction imposed by the node's ancestors, and $[begin(s)..end(s)] \subseteq [b..e]$, the restriction imposed by the current node. We put the pieces together and have $[min'..max'] := [min..max] \cap [b..e]$.

Case Node is `before` e.
Similarly, we do $[min'..max'] := [min..max] \cap [-\infty..e]$.

Case Node is `within` w.
As before, we have $[begin(s)..end(s)] \subseteq [min..max]$ from the ancestor nodes. Now, any (composite) event being generated in future event detections will have $end(s) \geq now$ (otherwise it has already been generated). It also has $end(s) - begin(s) \leq w$ by the current node's restriction. In summary $[begin(s)..end(s)] \subseteq [(now - w)..now]$, and this restriction applies also to all currently stored events if they will be used in constituting future composite events. Therefore, $[min'..max'] := [min..max] \cap [(now - w)..now]$.

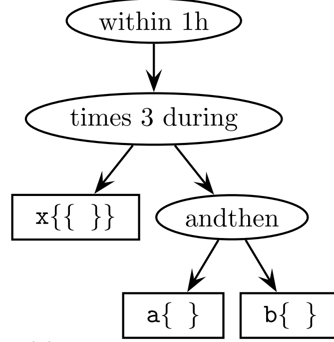
Case Node is `during [`b `..` e`]`.
As in the `in`-case: $[min'..max'] := [min..max] \cap [b..e]$.

Note that this deletion can decide whether or not to delete a stored event solely based on the current time and the event query. It does not depend on the received events.

Considering the received events could allow us to delete more events, e.g., in a query like the one of Figure 2.2 we could delete events in the left subtree of `times` (instances of `x{{ }}`) based on information about events in the right subtree: for each instance of `x{{ }}` there must be an earlier instance of `a{}` somewhere in the right subtree; otherwise `x{{ }}` cannot possibly satisfy the requirement to happen *during* an instance of `andthen [ a{}, b{} ]`. While considering received events in such a way can save memory, it would however complicate event deletion significantly. Whether the win in space consumption justifies the increased time consumption caused by the complication is doubtful. It generally seems preferable to work first with the simple event deletion and introduce such space-time trade-offs only when they are really necessary.

```
times 3 {
    x {{ }}
}
during  {
    andthen [ a{}, b{} ]
}
within 1h
```

(a) The event query



(b) The query's operator tree

Figure 2.2: Operator tree for a `times...during...`-query

Special Considerations. The preceding sections have provided us with the general ideas for event detection and event deletion. There are still some smaller details that have to be explained to be able to evaluate the full event query language as presented in Sections 2.2.3 and 2.2.4.

Duplicate Elimination. The evaluation algorithm discussed above can, in some rare cases, yield duplicate answers that should not be duplicated according to our formal semantics (Section 2.2.4). Consider evaluating the (admittedly somewhat pathological) event query

```
and {
  andthen[ a, b ],
  or  { a, b }
}
```

on the event stream $\mathcal{E} = \langle a^0, b^1 \rangle_0^1$ with the method outlined above. (For simplicity and conciseness we will use simple integers as time points and durations here and in the examples in the rest of the chapter.) Evaluation of the component `andthen [ a, b ]` will yield one answer ($s_{andthen} = \langle a^0, b^1 \rangle_0^1$). Evaluation of the component `or { a, b}` will yield two answers ($s_{or1} = \langle a^0 \rangle_0^0$ and $s_{or2} = \langle b^1 \rangle_1^1$). Accordingly, a join of the answers for evaluating the whole `and` will, without a duplicate elimination, yield two separate answers that are however equal: $s_1 = s_{andthen} \cup s_{or1} = \langle a^0, b^1 \rangle_0^1$ and $s_2 = s_{andthen} \cup s_{or2} = \langle a^0, b^1 \rangle_0^1$.

Our formal semantics is based on sets and relations and thus requires that the two answers are treated as one (note that here the queries have no free variables and thus the substitution sets are automatically equal). Rather than modifying the evaluation algorithms to catch cases as the one outlined above, it is easier to leave them as they are and to perform a duplicate elimination afterwards.

Equal answers are in particular equal in the ending time, so it suffices to perform duplicate elimination for each single query evaluation on an incoming event message. Also, it suffices to perform it only for the results delivered by the root node in the operator tree, or in case that events are counted (`times`).

Duplicate elimination can be performed in $O(n \log n)$-time (it can be reduced to sorting), where n is the number of answers delivered by the root node in a single query evaluation.

Generally, n will be very small (most times 0, in fact), so a naive $O(n^2)$ implementation of duplicate elimination (i.e., compare all tuples) might even be faster in practice. The naive implementation also avoids having to define an ordering relation on event sequences.

Partial Matches. An `andthen[[ ]]`-operator with a partial match specification returns as answer not only event instances matching the specified event queries, but also any atomic events happening between them. Therefore, the `andthen[[ ]]`-inner node needs access to these atomic events. Several possibilities exist; we choose here the following: extend it to have three children: a left child and a right child, both derived from an event query just as in the case of the total `andthen[ ]`, and an additional middle child that matches any incoming atomic event. The node has an event storage for each of its children in the form of a simple list (recall that only references to these atomic events need to be stored). Detection of events works the same way as in the total `andthen[ ]` case. The answer that is passed to the parent node is similarly extended: those events from the middle child's storage that happen between the events from the left and the right child are added to the answer's event sequence; the answer's substitution set is not modified.

Negation. Variables that occur both inside a `without`-construct and elsewhere in an event query (with a defining (negative) occurrence), need special consideration. Take, for example, the event query (using as mentioned above simple integers for time points)

```
andthen [
    without { a {var X} } during [0..1]
    b { var X },
]
```

being evaluated on $\mathcal{E} = \langle a\{0\}^0, a\{1\}^1, b\{2\}^2, b\{1\}^3 \rangle_2^0$.

According to the semantics (Section 2.2.4), the event query evaluates successfully upon reception of `b{2}`. The answer is $s = \langle b\{2\}^2 \rangle_0^2$, $\Sigma = \{\{X \mapsto 2\}\}$. Evaluation is however not successful upon reception of `b{1}`, since $X \mapsto 1$ would allow the negated `a{var X}` to evaluate successfully.

Keeping in mind that we want to use incremental evaluation, note that in this example the defining occurrence of X is part of a component (`b { var X }`) that will be evaluated only after the component with the non-defining occurrence (`without`) has been evaluated. Even more, if we replace `andthen [ ]` with `and { }` in the example, we can say nothing about their evaluation order. It is thus preferable to find a way to accommodate negation without relying on evaluation order, but rather using only the bottom-up data flow in the evaluation tree.

Our solution is to extend the notion of substitution sets. Up to now, substitutions only prescribe assignments of values to variables. We extend this, so substitutions can also "forbid" specific assignments. In the example above, the `without`-component would thus not prescribe any assignments (since there are no defining variable occurrences), but it would forbid assignment of 0 or 1 to X: $\Sigma_n = \{\sigma_n\}$ with $\sigma_n = \{X \not\mapsto 0, X \not\mapsto 1\}$.

A join of Σ_n with $\Sigma_{b1} = \{\{X \mapsto 2\}\}$ (from evaluation of `b{ var X }` at time point 2) in the evaluation of `andthen` will succeed with $\Sigma = \{\{X \mapsto 2\}\}$. A join of Σ with $\Sigma_{b2} = \{\{X \mapsto 1\}\}$ (from evaluation of `b{ var X }` at time point 3) in the evaluation of `andthen` will fail due to $X \not\mapsto 1$ and $X \mapsto 1$.

Forbidding certain assignments also integrates nicely with our alternative view on substitutions sets as constraints on the variables (cf. 2.2.5.3 above). The forbidden assignments

$\sigma_n = \{X \not\mapsto 0, X \not\mapsto 1\}$ simply give negated constraints $\neg(X = 0) \wedge \neg(X = 1)$. The implementation (see [30, Chapter 8]) is based on constraints rather than substitution sets. It is especially convenient to use constraints since the Xcerpt prototype implementation [64] provides a constraint solver we can rely on.

Existentially Quantified Variables. The language construct `any` in `times` *mult* `any var` $X_1, \ldots X_k$ $\{q\}$ `during` D and *mult* `of any var` $X_1, \ldots\ X_k$ $\{q_1, , \ldots, q_n\}$ `during` D (where D is an absolute time interval or a composite event query) introduces an existential quantification of the variables $X_1, \ldots X_k$. That is, answers to q or $q_1, \ldots q_n$ respectively need only agree in their substitutions on all variables *except* $X_1, \ldots X_k$. We cannot simply use the join operation $\bowtie$ from above (see Section 2.2.5.3) to build the answer to the composite event.

Let us analyze only the case `times` n `any var`$X_1, \ldots X_k$ $\{q\}$ `during` D first and state the problem clearer.

While D runs, q evaluates successfully a couple of times to answers (s_1, Σ_1), (s_2, Σ_2), $\ldots$, (s_N, Σ_N). These answers could be needed when D finishes to form an answer to the whole `times`-query and we store them.

Now, when D finishes, we need to find all subsets with cardinality n of the set $S := \{(s_1, \Sigma_1),$ (s_2, Σ_2), $\ldots$, $(s_N, \Sigma_N)\}$ which satisfy the conditions given by the semantics of the `times`-operator (cf. 2.2.4.3). These are (1) $\Sigma_i \mid_{V \setminus \{X_1, \ldots X_k\}} = \Sigma_j \mid_{V \setminus \{X_1, \ldots X_k\}}$ for all $1 \leq i < j \leq n$, (2) $\Sigma \subseteq \bigcup_{1 \leq n} \Sigma_i$, and (3) Σ_i maximal; Σ here denotes the substitution set of the answer of the whole composite event query.

Our first task is thus finding all subsets of S with cardinality n whose elements are equal on $\Sigma_i \mid_{V \setminus \{X_1, \ldots X_k\}}$. This is not hard, we can simply sort the elements (s_i, Σ_i) of S into buckets b_j according to their restricted substitution sets $\Sigma_i \mid_{V \setminus \{X_1, \ldots X_k\}}$. (Note that the substitution sets are maximal according to the condition (3), so we do not have to look at subsets of the substitution sets.) A bucket $b_j = \{(s_{j1}, \Sigma_{j1}), \ldots (s_{jn_j}, \Sigma_{jn_j})\}$ delivers an answer for the whole `times`-query if and only if it contains exactly n elements (i.e., $|b_j| = n_j = n$). Given such an answering bucket b_j, we simply set according to the second condition of the semantics: $\Sigma' := \bigcup_{1 \leq l \leq n_j} \Sigma_{jl}$.

If the duration D in the whole `times`-query is a composite event query q'', it has an answer (s'', Σ'') and we need to join Σ' and Σ'' before giving the result to the parent node in the operator tree: $\Sigma := \Sigma' \bowtie \Sigma''$. (Note that this can make Σ "smaller" than Σ'; hence the semantics only requires "$\subseteq$", not "$=$" in condition (2).) If D is an absolute time interval, simply set $\Sigma := \Sigma'$.

The extension of this idea to `times atmost` and `times atleast` is trivial. The extension to work for the `of` operator is also not hard: one simply has to modify the conditions on the bucket, so that its elements must have been generated by different queries.

2.2.5.4 Ideas for Optimizations.

The algorithm outlined up to now leaves much room for optimizations. We now discuss a few ideas how it could be optimized. Note that these are all only basic ideas that still require a lot of work to be worked out fully.

Stream-Based Evaluation of Atomic Event Queries. Event query evaluation has to evaluate a potentially high number of atomic event queries. Atomic event queries can be registered as stand-alone event queries or as constituting parts of composite event queries. While in a composite event query's operator tree inner nodes (implementing composition operators)

have to be evaluated only when their children deliver new events, the leaves (implementing atomic event queries) have to be evaluated every time a new event message is received. Since thus atomic event query evaluation plays such an important role also in composite event query evaluation, its optimization should usually provide the most benefits and be the most important.

We can formulate the optimization problem as follows: On a single incoming event message (XML document or data term) t we have to evaluate a number of atomic event queries (Xcerpt query terms) $q_1, \ldots, q_n$. Generally we can assume that the XML document t is small enough so it can fit into main memory.[16] The queries $q_1, \ldots, q_n$ are relatively static; only once in a while one of them will be removed or a new one added. In contrast, we receive new XML documents t in rapid succession and have to evaluate $q_1, \ldots, q_n$ on each.

Situations like this, where many queries are evaluated repeatedly on single XML documents (or data points, data tuples, etc.) have been analyzed in data stream processing [18]. In contrast to traditional database work, where usually the data is indexed, the goal in stream processing is to perform indexing *on the queries*; typically one tries to exploit similarities between different queries such as common sub-expressions, so that these are not evaluated for each query but only once.

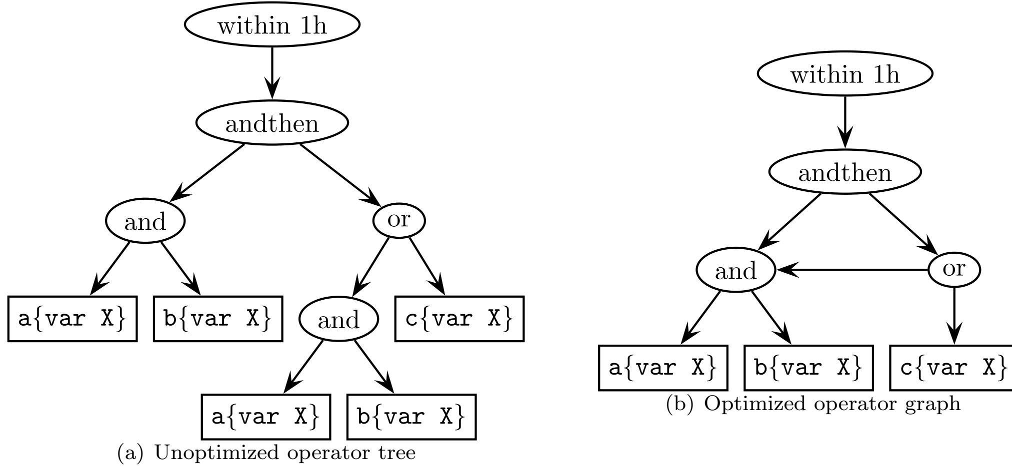
A stream-based evaluation of Xcerpt queries (i.e., atomic event queries) would provide a great optimization for the event query evaluation in XChange. Surely concepts from stream-based evaluation of other query languages, esp. XPath [7], can be borrowed for this. For a stream-based evaluation of Xcerpt queries in the framework of XChange we can however make some assumptions less common in existing work on stream-processing of XML:

- We only need to process single, whole XML documents one at a time, and can assume that each of these fits into main memory. This is in contrast to some other work in XML stream processing, e.g., SPEX [16, 55], where it is assumed that the stream itself is one large or even infinite XML document.
- Since we can read an incoming XML document completely before starting to evaluate queries on it, it can be worth considering to perform some (cheap) main-memory indexing operations on the XML document, say build a hash table for the labels. Preferably, any such data indexing should be such that it can be performed in one pass while parsing the XML document.
- Atomic event queries (Xcerpt query terms) allow an equivalent of joins inside a single event message (XML document) by using the same variable in different places. Many stream-processing systems explicitly forbid joins within a stream or allow only restricted forms. Note that joins between two different event messages are also possible in XChange, but only by means of composite event queries; thus there is no need to consider them here.

Shared Composite Event Detection. Similar to the way stream-based evaluation of atomic event queries above exploits shared sub-expressions in atomic event queries, one can try to exploit shared sub-expressions in composite event queries. This can be done both within a single composite event query, making the operator tree an acyclic graph, and among different composite event queries, leading to a sharing of (sub-)trees. A composite event detection system performing such optimization is EPS [53].

For example, the operator tree for the single composite event query

[16] Keep in mind that during the composite event query evaluation (processing of inner nodes in the operator tree) t needs not be accessed anymore; it suffices to maintain a reference to it when constructing the event



(a) Unoptimized operator tree

(b) Optimized operator graph

Figure 2.3: Optimization of an operator tree into an operator graph

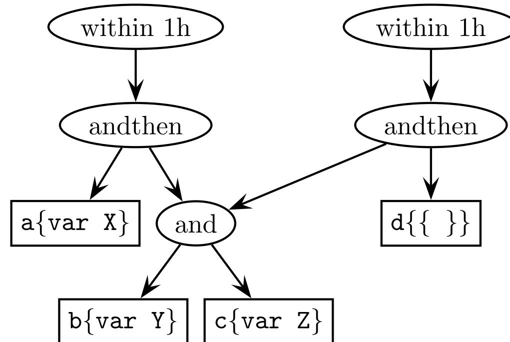


Figure 2.4: Optimization of composite event queries with shared sub-expressions

```
andthen [
    and {  a{var X}, b{var X}  },
    or {
        and {  a{var X}, b{var X}  },
        c{var X}
    }
] within 1h
```

depicted in Figure 2.3(a) can be optimized to the graph in Figure 2.3(b).
For another example, the composite event queries

sequences that are part of the answers. Thus t can be moved to secondary memory after the evaluation of all atomic event queries).

```
andthen [                                  andthen {
    a{ var X },                                and { b{var Y}, c{var Z} },
    and { b{var Y}, c{var Z} }       and       d{{ }}
] within 1h                                } within 1h
```

share the common sub-expression `and{ b{var X}, c{var Y}`. We can thus try to optimize their two operator trees into a graph such that they share a subtree for the common sub-expression as depicted in Figure 2.4.

When optimizing between different composite event queries, care has to be taken if the composite event queries were not registered at the same time. A common subtree of an earlier registered query can have memorized events prior to the registration of a later query; these events are not relevant for the query that has been registered later and need to be filtered out.

Inhibit or Delay Evaluation of Certain Subtrees. Some operators allow us to inhibit or delay evaluation of some of their subtrees. Consider the `andthen`-operator. As long as we have no answer for the left child, there is no need to evaluate the whole subtree of the right child at all; we can inhibit evaluation of the right subtree. Alternatively, we can delay evaluation of the left subtree as long as we have no answer from the right child, and only when we have an answer from the right child (which might not happen at all!) we start evaluating the left subtree; this requires that the incoming event messages for the left subtree are memorized while its evaluation is delayed.[17]

For the `andthen`-operator we cannot, of course, inhibit evaluation of the right subtree and delay evaluation of the left subtree both at the same time. We can however start by inhibiting evaluation of the right subtree until a first event in the left subtree has been detected and then switch to delayed evaluation of the left subtree.

Restructuring of Operator Trees. A common query optimization technique in databases is restructuring the initial operator tree to an equivalent, hopefully more efficient, operator tree. This relies on laws like commutativity, associativity, or distributivity for the operators of the query language.

In principle, restructuring of the operator tree as an optimization technique is also possible for XChange's event queries. For example, `and{ and{ and{a,b}, c }, d }` is equivalent to `and{ and{a,b}, and{c,d} }`.

We do face two problems however. First, there seem to be only a small number of laws that can be used for restructuring in XChange's event query language. The few obvious ones are associativity of the (binary) `and`, `or`, `andthen`, and distributivity of `and` and `or`. (Then again, maybe we just haven't found other valuable laws yet.) Second, we have no a priori information (e.g., a data dictionary containing statistics about data such as number of entries in a table) available that allows a good cost estimation. The only thing we can do is use statistics gathered in the past and try to "predict the future" with them and of course use simple heuristics that do not require any statistical information.

Still, there is one simple optimization that we can learn from relational databases that is very easy to do and should (if applicable) definitely pay off: push-selection. The `where`-clause in XChange gives conditions on the variables, much like a selection σ_C gives a condition C on data tuples. If it is used on a more complicated expression, it is often possible to push it into

[17] A language with support for lazy evaluation such as Haskell might actually perform such an optimization automatically without posing any work on the shoulders of the programmer.

the sub-expression. For example $\sigma_C(A \times B)$ can be rewritten to $\sigma_C(A) \times B$ if the condition C only applies to relation A. Similarly rewriting can be done with some event queries in XChange. For example:

```
andthen [                                andthen [
    and {                                    and {
        a{ var X },                              a{ var X },
        b{ var Y }              to               b{ var Y }
    },                                       } where { var X < var Y },
    c {var Z}                                c {var Z}
] where { var X < var Y } within 1h      ] within 1h
```

Push-selection might be a little less important in XChange than in relational databases: in XChange, the programmer of a query has the freedom to place the `where`-clause where he wants it; in relational databases, he is often restricted to Select-Project-Join (SPJ) queries. But push-selection is still a very valuable addition in XChange as it takes the responsibility of placing `where`-clauses efficiently from the programmer's shoulders.

Efficient Algorithms for Joins of Substitution Sets. Last but not least for an efficient composite event detection, efficient algorithms for each inner node that implements a composition operator need to be devised. Foremost, we have to investigate an efficient computation of the join $\Sigma_1 \bowtie \Sigma_2$ of substitution sets (or, equivalently the conjunction $C_1 \wedge C_2$ of two constraints in disjunctive normal form) since it is the most frequent operation. Strongly connected with this is the task to find good supporting data structures for the event storage in a node.

Our join of substitution sets is only a variant of a usual natural join that treats undefined values differently. Algorithms to efficiently compute joins have been investigated in the database community for a long time, and it should be no problem to adapt them. In contrast to the usual assumption made in databases, however, we assume in XChange that substitution sets are small enough to be kept in main memory. This gives a cost model that is quite different from those used in databases, so algorithms which are considered inefficient for normal secondary-storage databases might actually show good performance in XChange and vice versa.

2.3 XChange Conditions – Web Queries

Web queries are queries to persistent data (i.e. data of Web resources); they represent the "condition part" of XChange reactive rules (cf. Section 2.5). Web queries determine if certain conditions hold (e.g. a person making a rental order is one of the clients of the rental firm, or the notification of a flight cancellation concerns Mrs. Smith's flight) and gather data as variable bindings that is needed for performing the desired actions (e.g. insertion of a new client in the database with the information received through the rental order, or booking an overnight stay for Mrs. Smith where the flight date is used).

2.3.1 Web Queries with Xcerpt

In XChange, Web queries are expressed using the Web and Semantic Web query language Xcerpt [64] that is being developed in the Rewerse working group I4. An XChange Web query is an Xcerpt query, that is a negation, conjunction or disjunction of query terms. Query

terms are used here for specifying patterns for the data to be queried augmented with variables for selecting data of interest. Variables bindings can be restricted to given patterns. Non-structural conditions on variables are specified in a `where` clause attached to the query terms or the whole query.

Web queries (Xcerpt queries) can query persistent data directly or by querying views constructed by means of deductive rules (Xcerpt construct-query rules). A resource specification inside an Xcerpt query gives the Web resources to be queried. If no resource specification is given, the Xcerpt query is posed against the data constructed by means of the Xcerpt construct-query rules contained in the same XChange program; thus, complex querying problems can be elegantly solved by using views over multiple, heterogeneous data sources. Section 2.5.4 gives an example of an Xcerpt rule constructing such a view.

This section does not elaborate more on Web queries, as they have been developed as part of Xcerpt, within REWERSE WG I4. Section 2.5 shows how Web queries are used together with event queries and action specifications in XChange so as to give reactive rule specifications. For more information on the query language Xcerpt used for specifying Web queries see [64, 23, 21, 22, 65].

2.3.2 Semantics of Web Queries: Underlying Ideas

Given an XChange program P, the Web queries $\mathcal{Q}$ of P's reactive rules are Xcerpt queries, that is conjunctions (denoted $\texttt{and}\{Q_1\texttt{,}\dots\texttt{,}Q_n\}$, $Q_1 \wedge \dots \wedge Q_n$ or by $\bigwedge_{1\leq i\leq n} Q_i$), disjunctions (denoted $\texttt{or}\{Q_1\texttt{,}\dots\texttt{,}Q_n\}$, $Q_1 \vee \dots \vee Q_n$ or by $\bigvee_{1\leq i\leq n} Q_i$) or negation (denoted `not Q` or by $\neg Q$) of query terms of $\mathcal{T}^q$. Also, the deductive rules Dr_k of the form ${t_k}^c \leftarrow \mathcal{Q}_k$ are Xcerpt rules. The aim of P's deductive rules is to "provide" (inferred or transformed) data for the Web queries $\mathcal{Q}$. This section presents the underlying ideas of the declarative semantics of the query language Xcerpt and shows how this fits into the framework of XChange.

A model theory for the query language Xcerpt has been developed (see [64], Chapter 7), which follows the approach of classical Tarski-style semantics for first order logic. However, the distinctive features (such as the grouping constructs in the head of the rules and partial specifications of queries) of the language Xcerpt entailed considerable differences from the classical logic. Classical logic differentiates between *terms* (representing objects) and *atomic formulas* (representing statements about objects); though, Xcerpt terms are atomic formulas expressing the statement that the respective term exists. Informally, an *interpretation* is a set of data terms that specifies what data terms exist and a *model* is an interpretation containing the terms inferred by the given Xcerpt rules.

The model theory of Xcerpt considers Xcerpt programs (sets of Xcerpt rules) as *formulas*. Query, construct, and data terms, and $\bot$ (falsity) and $\top$ (truth) are constituents of atomic formulas. The connectives $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$, and the quantifiers $\forall$ and $\exists$ are used for constructing compound formulas. Atomic and compound formulas built this way are called *term formulas*. The formula representation of a set of rules $\{Dr_1, \dots, Dr_p\}$ is the conjunction of the formula representation of each Dr_k and of the data terms that represent the specified resources as internalised (data terms are considered part of the program). The grouping constructs need special treatment and, thus, symbols $\ll \cdot \gg$ are used for denoting the scope of all grouping constructs contained in the rules.

Example 2.33 (Formula Representation of Xcerpt Rules) *Consider the following set of Xcerpt rules:*

```
a{all var X, var Y} ← and{ b{{var X}}, c{{ d{var X,var Y} }} }
b[ var X ]          ← c{{ d[var X] }}
c[ d[e,f], d[g,h] ]
```

This set of rules is represented as a formula as follows:

```
∀ Y ≪a{all var X, var Y} ← b{{var X}} ∧ c{{ d{var X,var Y} }} ≫ ∧
∀ X ≪b[ var X ]  ← c{{ d[var X] }} ≫ ∧ c[ d[e,f], d[g,h] ]
```

Interpretations. An interpretation is a tuple $M = (I, \Sigma)$: I is a set of data terms of $\mathcal{T}^d$ and $\Sigma \neq \emptyset$ is a grounding substitution set, i.e. provides assignments for all variables with at least one defining occurrence in the formulas considered.

Satisfaction and Models. An atomic formula F is considered to be satisfied in interpretation M if and only if its ground instance (obtained by applying the substitutions of Σ to F) simulates into a term of I. The satisfaction of a term formula (i.e. atomic or compound formula) is defined recursively over its structure. The following definition is taken from [64], Section 7.4.2, on pages 151-152:

Definition 2.1 (Satisfaction, Model)

1. *Let $M = (I, \Sigma)$ be an interpretation (i.e. a set of data terms I and a substitution set Σ), and let t be a construct or query term.*

   The satisfaction of a term formula F in M, denoted by $M \models F$, is defined recursively over the structure of F:

$M \models \top$		*holds*
$M \models \bot$		*does not hold*
$M \models t$	*iff*	*for all $t' \in \Sigma(t)$ there exists a term $t^d \in I$ such that $t' \preceq t^d$*
$M \models \neg F$	*iff*	$M \not\models F$
$M \models F_1 \wedge \cdots \wedge F_n$	*iff*	$M \models F_1$ *and ... and* $M \models F_n$
$M \models F_1 \vee \cdots \vee F_n$	*iff*	$M \models F_1$ *or ... or* $M \models F_n$
$M \models F \Rightarrow G$	*iff*	$M \models \neg F \vee G$
$M \models \forall x.F$	*iff*	*for all $t \in I$ holds that $M' = (I, \Sigma') \models F$, where $\Sigma' = \left\{\sigma \circ \{x \mapsto t\} \mid \sigma \in \Sigma\right\}$*
$M \models \exists x.F$	*iff*	*there exists a $t \in I$ such that $M' = (I, \Sigma') \models F$, where $\Sigma' = \left\{\sigma \circ \{x \mapsto t\} \mid \sigma \in \Sigma\right\}$*
$M \models \forall^* \ll t^c \leftarrow Q \gg$	*iff*	*$M' = (I, \Sigma') \models t^c$ for a maximal grounding substitution set Σ' for Q with $M' \models Q$*

2. *If a formula F is satisfied in an interpretation $\mathcal{M}$, i.e. $\mathcal{M} \models F$, then $\mathcal{M}$ is called a* model *of F.*

Note that in the definition above, $\forall^*$ is used to universally quantify all free variables in a formula.

Given an Xcerpt program Pr, a model for Pr is an interpretation (I, Σ) where I contains *all* data terms that are inferred (or produced) by the rules of Pr. *Note* that I may contain also data terms that are unrelated to Pr. A concrete example for satisfaction of Xcerpt programs is given in [64], Chapter 7, pages 152-153. However, the focus in this section is on the parts

of XChange programs that are expressed in Xcerpt. Recall that an XChange program has the form $P = \{Rr_1, \dots, Rr_m, Tr_1, \dots, Tr_n, Dr_1, \dots, Dr_p\}$, where Rr_i, $1 \le i \le m$ and Tr_j, $1 \le j \le n$ are reactive rules that may have "condition parts" specified by Xcerpt queries. Also, Dr_k, $1 \le k \le p$ are Xcerpt rules. Consider $\mathcal{Q}_l$, $1 \le l \le m+n$, the Xcerpt queries associated with the reactive rules of P. The interest is on the satisfaction of the formulas of the form:

$$\mathcal{Q}_l \wedge \bigwedge_{1 \le k \le m+n} \forall^* \ll t_k^c \leftarrow Q_k^r \gg \wedge d_h \quad , \quad 1 \le l \le m+n \quad ,$$

,
where Dr_k is of the form $\quad t_k^c \leftarrow Q_k^r \quad$, for $1 \le k \le p$ and $d_h \in \mathcal{T}^d$ are data terms that represent the internalised Web resources that are specified in the queries. *Note* that the interest for XChange programs is not on the conjunction $\bigwedge_{1 \le l \le m+n}$ of the formulas given above. As Definition 2.1 covers also formulas of the form given above, the semantics of the "condition parts" of XChange reactive rules is given by the model-theoretic approach of Xcerpt.

Example 2.34 (Formula Representation of Condition Parts and Deductive Rules) *Consider an XChange program P that consists of the following rules:*

```
<transaction_spec> ←r a {{ var Z }} ←r <event_query>
a{all var X, var Y} ← and{ b{{var X}}, c{{ d{var X,var Y} }} }
b[ var X ]            ← c{{ d[var X] }}
c[ d[e,f], d[g,h] ]
```

Note that P contains one transaction rule whose event query and transaction specification are not specified as they do not contribute to the formula representation of Web queries and deductive rules. The deductive rules of P are the Xcerpt rules of Example 2.33 with a slight modification of the data term. The above "condition part" and set of deductive rules is represented as formula as follows:

```
a {{ var Z }} ∧
∀ Y ≪a{all var X, var Y} ← b{{var X}} ∧
c{{ d{var X,var Y} }} ≫ ∧
∀ X ≪b[ var X ] ← c{{ d[var X] }} ≫ ∧
c[ d[e,f], d[g,f] ]
```

A fixpoint semantics for Xcerpt programs without negation is proposed in [64, Chapter 7, Section 5]. A fixpoint operator T_P is defined by applications of which a fixpoint for Xcerpt programs is iteratively constructed. It is proved that the fixpoint of a program is also a model of the program [64, Theorem 7.10 on page 155].

2.3.3 Evaluation of Web Queries: Basic Ideas

Evaluating the Web queries given as "condition part" of XChange reactive rules presupposes evaluating Xcerpt queries against specified resources and chaining of Xcerpt rules so as to evaluate Xcerpt queries against views constructed by means of Xcerpt rules. The operational semantics of the query language Xcerpt is defined in [64], Chapter 8. The underlying ideas of the semantics are shortly described in this section.

An algorithm is defined for evaluating Xcerpt programs based on two parts: an algorithm called *simulation unification* is given and a *backward chaining* algorithm that uses simulation

unification. The evaluation is based on a simple constraint solver that applies simplification rules to a constraint store consisting of conjunctions and disjunctions of constraints. An example of a constraint is a *simulation constraint* expressing e.g. possible bindings for a variable. The constraint store yields bindings for the variables occurring in Xcerpt query and construct terms.

Simulation unification takes two terms and returns a set of variable substitutions (called *simulation unifier*) such that their applications to the terms make them simulate one into another. A proof is given for the soundness and completeness of the simulation unification algorithm in [64].

The backward chaining algorithm used in evaluating Xcerpt programs is inspired by the SLD resolution calculus used in logic programming [46]. It is shown that the algorithm is sound with respect to the fixpoint semantics developed for Xcerpt; also a (weak) completeness of the algorithm (is complete in cases where the algorithm terminates) is proved. Criteria for termination are also described in [64].

2.4 XChange Actions

2.4.1 Update Patterns

Existing proposals for update languages for the Web rely on a path-expression-matching operation that selects nodes within the input document; the selected nodes are the target of the update operations. XChange follows *another approach* for updating data – update specifications are *patterns* for the data to be updated augmented with the desired update operations. By following a pattern-based approach for XChange updates,

- the specification of desired updates is simple and intuitive, as an update specification is like a form where data needs to be inserted, replaced, or deleted;
- the whole language XChange follows a single approach – a pattern-based one – minimising thus the effort of learning the language; programmers need only understand the concept of data patterns.

Recall that only persistent data (i.e. data of Web resources) can be updated, volatile data can not; thus, when talking about updating data, only updates to persistent data are meant. (*Note* that the short notation used in the previous section where the envelope of events has been left out is not adopted here; all examples given in this section use persistent data!) XChange has been primarily developed for updating XML data, this includes also any data format having an XML serialisation (such as RDF data). However, XML data represents data trees while e.g. RDF data represent graphs. XChange can be used for updating graph data, but one needs to decide on the semantics of updates on graph data and to modify the update execution accordingly. A discussion on updating graph data is given later in this section. In the XChange framework, XML data has a more compact representation as data terms; thus, XChange updates are patterns for data terms to be modified. Data terms are either *local* – on the machine the XChange program is running on – , or *remote* – on another machine (where possibly another XChange program is running on) at a given Web resource.

An XChange update specification contains a *resource specification* (i.e. Web resources whose data are to be modified) and an *update pattern* (i.e. gives information about how data is to be updated). The grammar rules defining XChange (elementary) updates are:

```
E_Update   ::= "in" "{" "resource" "{" Res_Spec ("," Res_Spec)* "}" ","
                        U_Term  "}"
Res_Spec ::= Uri ("," Format)?
```

`Uri` is the URI of the resource on the Web; the optional `Format` specifies the format (e.g. XML, HTML, RDF) of data found at `Uri` and may be used by the runtime system to choose the correct parser. When more than one Web resource is given in an update specification, the specified update pattern is used for updating each of them.

The outline of this section is as follows: Section 2.4.1.1 introduces the notion of *update term* and discusses general characteristics of update operations in XChange. The sections that follow discuss in more detail the three kinds of update operations offered by XChange. The discussion on insertion (Section 2.4.1.2), deletion (Section 2.4.1.3), and replacement (Section 2.4.1.4) specifications applies to updating subterms of data terms; the special case of updating the root of a data term is discussed in Section 2.4.1.5. Discussions on the chosen approaches to updating data are offered throughout the whole section.

2.4.1.1 Update Terms

Developing the update language of XChange has shown that patterns are amenable to specifying updates to Web data. For specifying updates that are to be executed, the language XChange offers a special kind of patterns, called *update terms*, which are introduced in this section. This is consistent with the pattern-based approach that has been followed in designing XChange.

Definition 2.2 (Update Term) *An update term is an Xcerpt query term augmented with the desired update operations. The query term gives a pattern for the data to be modified. The update operations may be insertions, deletions, or replacements of data. An insertion operation specifies an Xcerpt construct term that is to be inserted, a deletion operation specifies an Xcerpt query term for deleting all data terms matching it, and a replace operation specifies an Xcerpt query term to determine data terms to be modified and an Xcerpt construct term for their new values.*

A more detailed discussion on XChange update operations is given in the subsequent sections. More than one update operation can be specified in an update term. Moreover, different kinds of update operations (insertions, deletions, and replacements) can be specified in an update term. Update operations cannot be nested (this will be clear after the update operations are introduced). Thus, an XChange update term is defined through the following grammar rules:

```
U_Term ::= Upattern
       | "desc" Upattern
       | U_Root

Upattern  ::= Label "{" UQ_List "}"
          |  Label "{{" UQ_List "}}"
          |  Label  "[" UQ_List "]"
          |  Label "[[" UQ_List "]]"

Label   ::= label | "var" Var_Name UQ_List ::= ((Query_Term |
```

```
U_Term) ",")* U_Term ("," (Query_Term | U_Term) )*
        | U_Op
U_Op    ::= Ins_Op
        | Del_Op
        | Rep_Op
```

Lifting (removing) all update operations from an update term produces a query term. This query term will be called in the following the *subjacent* query term of the corresponding update term. For obtaining an update term, update operations are specified in a (subjacent) query term wherever a query subterm can be specified. The subjacent query term of an update term

(a) specifies a pattern for the data to be modified (*note* that the pattern is applied to the data before any updates are performed); only documents whose representation as data term matches the query are modified (i.e. no updates are executed if simulation unification of the subjacent query term with the data term to be modified fails);

(b) generates bindings for the specified variables that are to be used in the execution of update operations specified in the update term.

The update operations specified in an update term

(a) specify what kind of updates to execute, i.e. insertions, deletions, or replacements;

(b) specify which part of the data is to be modified (e.g. where to insert new data), given by the position of update operations inside an update term;

(c) use variables bound in the subjacent query term, the event part (event query), the condition part (Xcerpt query) of the XChange rule whose action contains the updates to be executed. Update operations might use also new variables, i.e. variables not occurring in the subjacent query term, event part, or condition part.

2.4.1.2 Insertion Specification

For inserting new data into a data term, one has first to construct the data (terms) to be inserted. Thus, an XChange insertion specification contains always an Xcerpt construct term, i.e. a pattern that makes use of variables to construct new data terms. Where the new data is to be inserted is given either implicitly (by the position of the update operation inside the update term) or explicitly (e.g. by explicitly giving the position at which the new data is to be found after the insertion is performed). The keyword `insert` is used for specifying insertion operations in XChange.

The grammar rules defining XChange insertion operations are given next (the constructs are explained in this section through simple examples):

```
Ins_Op ::= "insert"  Construct_Term
       |  Order_Ins
       | "insert"  Mult_Term

Order_Ins  :: =  "after"  Query_Term    "insert"  Construct_Term
           | "insert"  Construct_Term   "before"  Query_Term
```

```
           | "at"  Position  "insert"   Construct_Term

Mult_CTerm ::= "all"    Construct_Term  (Order)?
           | "some"  Nr  Construct_Term  (Order)?
           | "first" Nr  Construct_Term   Order

Order      :: = "order by" ( AD )? ( LN )?  "[" Vars "]"
AD ::= "ascending" | "descending"
LN   ::= "lexical" | "numerical"
```

The subjacent query term of an update term containing insertion specifications is obtained from the update term by removing the keywords `insert` and the specifications of the new data to be constructed that follow `insert`; positions specifications (of the form *at Position*) and keywords `before` and `after` are also removed. The obtained specification is a query term that is to match the data term to be modified.

Example 2.35 (Simple XChange Update Term Specifying Insertion (1)) *The following update term specifies insertion of data term $d\{e\{\}, f\{\}\}$ in a data term that matches the subjacent query term $a\{\{b\{\{\}\}, h\{\{\}\}\}\}$.*

```
a {{
  b {{ }},
  insert d { e{}, f{} },
  h {{ }}
}}
```

The new data is to be inserted as a child of the root (labelled a). No exact position of the new child is given; curly braces denote also that the order of child elements is not important. Consider now the following two data terms: the result of applying the above update term to the data term on the left yields the data term on the right. The data inserted is written in blue.

```
a {
   b { i{}, j{} },
   h { k{} }
}

   Before update
```

```
a {
   b { i{}, j{} },
   d { e{}, f{} }
   h { k{} }
}
   After update
```

Variables can be used inside insertion specifications; they are bound in the subjacent query, the event query, and/or the Xcerpt query of a rule and used for constructing the data to be inserted. The bindings for the variables are nondeterministically chosen from the set of variable substitutions resulting from evaluating the subjacent query term and the other parts of the associated rule. Recall that Xcerpt construct terms may contain grouping constructs (`all`, `some`) for gathering *all* or *some* of the variable bindings. One may also use insertion specifications like *insert var V* $\rightarrow$ *Construct_Term*; the variable is bound to the data term constructed with *Construct_Term* and can be further used in other update operations of the same update term.

Example 2.36 (Simple XChange Update Terms Specifying Insertion (2)) *The following update term specifies insertion of a data term that is constructed from $d\{var\ X\}$ in a data*

term that matches the subjacent query term $a\{\{var\ X\}\}$. The variable binding that is used to construct the data term to be inserted is nondeterministically chosen from the set of bindings that results from matching the subjacent query term with the data term to be modified.

```
a {{
  var X,
  insert d { var X }
}}
```

Consider the following two data terms. The one on the left hand side is to be modified by the above update term. The result of simulation unifying the subjacent query term with the data term to be updated is the substitution set $\Sigma = \{\{X \mapsto b\{i\{\}, j\{\}\}, \{X \mapsto h\{k\{\}\}\}\}$. Applying the update term to the left data term, might result in the right data term. (Since the substitution set contains two substitutions for X, two possible results are conceivable.) Again, new data is written in blue.

```
a {
    b { i{}, j{} },
    h { k{} }
}

    Before update
```

```
a {
    b { i{}, j{} },
    d {
       h { k{} }
    }
    h { k{} }
}
After update
```

Consider now a slight modification of the above update term; a d-labelled element containing all bindings for X is to be inserted. The update term is shown in the following example on the left side; the result of applying the update term to the data term given above on the left is shown in the following on the right hand side.

```
a {{
   var X,
   insert d { all var X }
}}
```

```
a {
   b { i{}, j{} },
   h { k{} },
   d {
        h { k{} },
        b { i{}, j{} }
   }
}
```

Position Specification. When updating ordered data (i.e. documents where the order of the elements is important), it is necessary to have means for inserting new data at a certain position in the data terms to be modified. The position can be specified relatively to existing subterms (i.e. before or after subterms matching a given query term) or explicitly through integers denoting the subterm position relative to its parent term.

Example 2.37 (Simple XChange Update Terms Specifying Insertion (3)) *Suppose now that one needs to introduce c-labelled elements (right) after each b-labelled element child of the root (i.e. as next sibling node in the tree representation of data); assume that the data term to be modified is ordered. Consider the following update term (bindings for the variable X are obtained from the other parts of the rule having the update term as action):*

```
a [[
   b {{ }},
   insert c { var X }
]]
```

This update term does not have the desired effect. A sample data term is given next; consider the substitution set $\Sigma = \{\{X = \text{"}content\text{"}\}, \{X = \text{"}more\ content\text{"}\}\}$. *The data term before the update is shown on the left, after the update on the right.*

```
a [                                a [
   b { i{} },                         b { i{} },
   h { k{} },                         h { k{} },
   d {},                              c { "more content" },
   b {}                               d {},
]                                     b {},
                                      c { "content" }
                                   ]

   Before update                      After update
```

As the above data terms show, the given update term can be used for inserting c-labelled nodes after b-labelled ones, but they are not necessarily next sibling nodes in the result. Thus, XChange offers also insertions of the form $after$ *Query_Term insert Construct_Term, meaning that after each data term matching the query term, a new data term is introduced as the next element in the document order (the next sibling node in the tree representation of the data). The desired effect can be obtained by using*

```
a [[
   after b {{ }} insert c { var X }
]]
```

In cases where the whole structure of the data to be modified is not known, the insertion after can not be used for inserting new data before subterms matching a given query term. Thus, XChange offers also a `before` counterpart of the insertion `after`. The insertion specification has the form *insert Construct_Term* $before$ *Query_Term*, meaning that before each data term matching the query term, a new data term is introduced as the previous element in the document order (the nodes will be next sibling nodes in the tree representation of the data). *Note* that the position of the construct term in the `after` and `before` insertion specifications denote the position of the new data relative to the subterms matching the given query term.

XChange offers support for introducing new data terms at a given position in the data term to be modified. Position gives the position of the inserted subterm below its parent term. Given a parent term in a data term, the first subterm below it has position 1, the last one position -1. In an insertion specification, the position is either a positive integer or a negative integer (variables can not be used instead of position specification inside an insert operation). Thus, inserting new data at position -1 means insertion as the last subterm of the given parent term. Consider the insertion of a new (sub)term *new* at position p below a parent b into a data term where the desired position is taken by a subterm *old*; after the update, the modified data term contains below b the subterms *new* at position p, *old* at position $p+1$, and the subterms following *old* at position $old_position + 1$, where *old_positon* is the position before the insertion.

Example 2.38 (Simple XChange Update Terms Specifying Insertion (4)) *The following update term inserts at last position in a catalogue a discount of* 10 *percent for all products not of type* `New Arrival`*.*

```
catalogue[[
   product {{
      id { var PId },
      without type { "New Arrival" }
   }},
   at -1 insert discount {
                              products { all var PId },
                              percent { "10" }
                         }
]]
```

Multiplicity Specification. With the insertion constructs exemplified until now, one can not make insertions of the form *insert all var Product*, as *all var Product* is not an Xcerpt construct term. But such kind of insertions are useful in practice. Moreover, one might need not only to insert all data terms constructed from a given construct term, but perhaps just some of these data terms (possibly chosen by means of a certain criterion). For this, the grouping constructs `all` and `some` of Xcerpt are used; recall that they gather *all* and *some*, respectively, possible instances of the construct term they precede. An order of the data terms to be inserted can be specified by means of the `order by` construct followed by a criteria and list of variables; the order is determined by the given criteria "applied" on the values of these variable bindings. (A detailed explanation of the grouping constructs can be found in [64], Section 4.6.2, pages 93 – 99.)

Example 2.39 (Simple XChange Update Terms Specifying Insertion (5)) *At some universities (e.g. the national universities in Romania) the best students are "awarded" by receiving a studentship for the next teaching term. The following update term is used to introduce in a data term the five best students (best in terms of their final grade) per teaching unit, for the ended teaching term.*

```
BestResults {{
   TeachingUnit [[
      insert var Term,
      UnitName { var N },
     insert first 5 var Stud order by [ var Grade ]
   ]]
}}
```

The students are ordered by their final grade obtained for the ended teaching term. The bindings for the variables are obtained by evaluating the next Xcerpt query; it represents the condition part of an XChange rule having in the action part the update term given above.

```
in { or { resource {"file:ai.xml", "file:net.xml", "file:db.xml"}
},
   desc TeachingUnit [[
```

```
            Name { var N },
            var Term → TeachingTerm {{ }},
            Students [[
               var Stud → Student {{
                              FinalGrade { var Grade }
                         }}
            ]]
         ]]
}
```

Note that XChange insertions have no duplicate elimination semantics, i.e. the subterms constructed with the given construct term are inserted regardless whether they already exist in the data term to be modified or not. For example, applying an update term like

```
a {{
   var X,
   insert all var X  }}
```

to a data term results in "doubling" the children of the root.

2.4.1.3 Deletion Specification

In order to delete parts of an XML document one has to specify a (possibly incomplete) pattern for the data to be deleted. The keyword `delete` is placed before these query patterns for specifying deletion operations in XChange. The informal meaning of a deletion operation of the form *delete Query_Term* can be resumed to (a) *all* (sub)terms matching *Query_Term* are to be deleted, and (b) the *whole* (sub)term (the whole subtree, in the tree representation of the data) matching *Query_Term* is to be deleted.

Similar to insertion specifications, the data term to be updated needs to match a given query term – the subjacent query term of the update term. Consider an update term of the following form (only delete operations are considered for simplicity) $label\{\{q_1, q_2, ..., q_n, delete\ q_{n+1}, q_{n+2}, ..., q_m\}\}$. The subjacent query term for the given update term may be (obtained by leaving the keyword `delete` out)

$$label\{\{q_1, q_2, ..., q_n, q_{n+1}, q_{n+2}, ..., q_m\}\}$$

or $label\{\{q_1, q_2, ..., q_n, q_{n+2}, ..., q_m\}\}$ (obtained by leaving the whole delete operation out). The first approach is considered in XChange – the subjacent query term of an update term containing only delete operations is obtained by leaving the keywords `delete` out – in the example above, $label\{\{q_1, q_2, ..., q_n, q_{n+1}, q_{n+2}, ..., q_m\}\}$. The rationale behind it is that for deleting subterms of a data term, these subterms need to exist, that is the data term to be modified needs to contain subterms matching the query terms following the `delete` keyword.

The grammar rules defining XChange's deletion operations are given next:

```
Del_Op     ::= "delete"  Query_Term
           |   "delete"  Mult_QTerm

Mult_QTerm ::= "some"  Nr  Query_Term
           |  "first"  Nr  Query_Term  (Order)?
```

Example 2.40 (Simple XChange Update Term Specifying Deletion (1)) *Assume that a data term is to be modified that matches the query term given below on the left. All c-labelled subelements of a-labelled elements having at least one subelement labelled b are to be deleted. The update term given on the right specifies the deletion. Note how easy the desired delete operation can be specified: one only needs to put the keyword* `delete` *in front of the query term that matches the data terms to be deleted.*

```
desc a {{
       b {{ }},
       c {{ }}
}}
```

```
desc a {{
       b{{ }},
       delete c {{ }}
}}
```

Example 2.41 (Simple XChange Update Term Specifying Deletion (2)) *The following update term deletes all subterms labelled b, regardless of their depth within the a-labelled term representing the whole document to be modified. Note that only b-labelled subterms are deleted, their parents (more concrete, their ancestors) remain in the data term if they do not have the label b.*

```
a {{
 delete desc b {{ }}
}}
```

For inserting data terms at a given position, a construct is offered in XChange; for deleting data terms having a given position and matching a given query term, no extra construct is needed. This can be specified inside the query term by means of the construct `position` of the language Xcerpt; a query term of the form *position Pos q* matches those data terms t found at position Pos in the queried data term, where $q \preceq t$. The position specification Pos is either a positive integer (where 1 is the position of the first subterm of a parent), a negative integer (where -1 is the position of the last subterm), or a variable that matches with the position of subterm (matching q) and binds to it as a positive integer (cf. [64], Section 4.3.3, pages 73-74).

The update operations exemplified until now have as effect the deletion of *all* subterms of a data term matching a given query term. By using Xcerpt's position specification, subterms with a given position below its parent term can be deleted (cf. above). However, one has no means for specifying deletion of a given number of subterms, possibly chosen by using a given criteria. Thus, two forms of deletion specifications are offered: A delete specification *delete some Nr Query_Term* (with Nr positive integer, $1 \leq Nr$) specifies deletion of n subterms matching *Query_Term*, where n is the maximal number of such subterms with $n \leq Nr$. A delete specification *delete* $first$ *Nr Query_Term order by* (*Criteria*) [$Variable_List$] specifies deletion of the first (regarding the order given by the specified criteria "applied" to the bindings for the variables of $Variable_List$) m subterms matching *Query_Term*, where m is the maximal number of such subterms with $m \leq Nr$. The order specification can be left out for deleting subterms of an ordered term; *delete* $first$ *Nr Query_Term* specifies deletion of the first (taking the document order into account) m subterms matching *Query_Term*.

Example 2.42 (Simple XChange Update Term Specifying Deletion (3)) *The following update term deletes maximal two subterms of the root element; they match* $b\{\{varX, varY\}\}$, *where the lexical order on the values of variable X determine which are the two terms to be deleted.*

```
a {{
 delete first 2  b {{ var X, var Y }} order by lexical [ var X ]
}}
```

Assume that the following substitutions for the variables are obtained by evaluating the other parts of the XChange rule having the above update term as action part: $\{X = \text{"}cde\text{"}\}$ *or* $\{X = \text{"}abc\text{"}\}$ *or* $\{X = \text{"}abf\text{"}\}$ *and* $\{Y = i\{\}\}$ *or* $\{Y = f\{\text{"}g\text{"}\}\}$. *The data term to be modified is given next on the left hand side, while the data term after the deletion has been performed is given next on the right hand side.*

```
a {                                a {
   b { i {}, "cde" },                 b { i{}, "cde" },
   h { k{} },                         h { k{} },
   b { f {}, "abf" },                 b { f {}, "abf"},
   b { f {"g"}, "abc"},               j { f { k {} } }
   j { f { k {} }}                 }
}
   Before update                      After update
```

2.4.1.4 Replace Specification

To specify a replace operation, one has to specify a (possibly incomplete) pattern for the data that is to be modified and a (complete) pattern for the new data to be found instead. The keyword `replaceby` is used in XChange for expressing a replace operation.

The grammar rules defining XChange replacement operations are given next:

```
Rep_Op ::= QTerm  "replaceby"  CTerm

QTerm  ::= Query_Term
       |  Mult_QTerm
CTerm  ::= Construct_Term
       |  Mult_CTerm
```

Consider an update term of the following form (only replacement operations are considered for simplicity) $label\{\{q_1, q_2, ..., q_n, q_{n+1}$ *replaceby ct* $, q_{n+2}, ..., q_m\}\}$. The subjacent query term for the given update term is $label\{\{q_1, q_2, ..., q_n, q_{n+1}, q_{n+2}, ..., q_m\}\}$; it is obtained by omitting the keyword `replaceby` and the specification of the new data to be found instead of q_n. Multiplicity specifications are also left out for obtaining the subjacent query term.

The informal meaning of a replacement operation of the form *Query_Term replaceby Construct_Term* can be resumed to (a) all terms matching *Query_Term* are to be replaced, and (b) each such term is to be replaced by a data term constructed with *Construct_Term*. (The constructed data term used for replacement is chosen in the same manner as for insertions.) Multiplicity specifications (`all`, `some Number`, and `first Number`) for the *Construct_Term* express that (instead of a single term) a set of terms (containing all, some or first *Number* constructed terms) is used for replacing each term t, where $Query_Term \preceq t$. Multiplicity specifications (`some Number` and `first Number`) for *Query_Term* express that a given number of (instead of all) terms t are to be replaced, where $Query_Term \preceq t$. Multiplicity specifications for *Query_Term* and for *Construct_Term* can be combined (as the above grammar rules show);

the resulting replace operations specify that each term of a set of terms is to be replaced by a set of constructed terms.

Proposition 2.1 (Necessity of Replacement Operation) *A replace operation of the form Q_Term replaceby C_Term has not the same effect as the sequence of two update operations of the form delete Q_Term and insert C_Term, where Q_Term, C_Term are specifications of (possibly) multiplicity followed by a query term and a construct term, respectively.*

Cf. Proposition 2.1, XChange's replace operation is not syntactic sugar for a delete and an insert operations performed in sequence. The next, simple example intends to clarify the effect of a replace operation in XChange and motivates the previous statement.

Example 2.43 (Simple XChange Update Term Explaining the Replace Operation) *The example gives an update term that specifies the replacement of each binding for the variable X with a binding for the variable Y.*

```
label {{
    var X replaceby var Y
}}
```

The same binding for Y is used to replace the data terms that are bindings for X. If the XML document that is queried for bindings for Y is not ordered, the first binding for Y found in the evaluation process of the corresponding query is used. If this XML document is ordered, the first, taken the document order in consideration, binding for Y is used for replacement. Some explanations follow for clarifying the effect of such a replace operation. Bindings for Y are provided by evaluating the event query and/or Xcerpt query of the XChange rule having the update term as its head (action part). The following cases for the above replace example can be distinguished:

- *Only one binding for the variable X and only one binding for the variable Y are returned from the evaluation of the XChange rule containing the update term.*

  Let $\{X = x_1\}$ and $\{Y = y_1\}$ be the obtained bindings. The update var X replaceby var Y has the effect of delete x_1 and insert y_1 (instead).

- *More than one binding for X and only one binding for Y are obtained from the evaluation of the XChange rule containing the update term.*

  Let $\{X = x_1\}$ or $\{X = x_2\}$ or $\ldots \{X = x_n\}$ (with $1 \leq n$) be the bindings for X and $\{Y = y_1\}$ the binding for Y. The update var X replaceby var Y has the effect of delete x_1 and insert y_1 (instead) AND delete x_2 and insert y_1 (instead) AND ... AND delete x_n and insert y_1 (instead).

- *More than one binding for X and more than one bindings for Y are obtained from the evaluation of the XChange rule containing the update term.*

  Let $\{X = x_1\}$ or $\{X = x_2\}$ or $\ldots \{X = x_n\}$ (with $1 \leq n$) be the bindings for X and $\{Y = y_1\}$ or $\{Y = y_2\}$ or $\ldots \{Y = y_m\}$ (with $1 \leq m$) be the bindings for Y. The update var X replaceby var Y has the effect of delete x_1 and insert y_1 (instead) AND delete x_2 and insert y_1 (instead) AND ... AND delete x_n and insert y_1 (instead) (y_1 is chosen as explained above).

- *More than one binding for X is obtained from the evaluation, but no binding for Y.*

  In this case, one possibility for performing the update var X replaceby var Y would be to delete all bindings for the variable X, because there is nothing to be inserted instead. A replace operation is to be understood as an atomic operation, in the sense that one can not split it into a delete operation followed by an insert operation. Though, the effect of a replace operation is sometimes the same as a delete operation followed by a well chosen insert operation (as in the previous cases). Thus, in XChange an update var X replaceby var Y has no effect (i.e., the data to be updated remains unchanged) in the case that the evaluation of the XChange rule containing it does not return bindings for Y.

- *There is no binding for X, but more than one binding for Y is obtained from the evaluation.*

  Using the same arguments as in the previous case, the update operation var X replaceby var Y has no effect on the data to be updated if no bindings for X are returned from the evaluation of the XChange rule containing the replace update.

- *No binding for X and no binding for Y is obtained from the evaluation. In this case the replace operation has no effect on the data to be modified.*

Example 2.44 (XChange Update Term Converting Prices From Euro to Dollar) *The example gives an XChange update term that specifies the modification of the used currency from euro to US Dollar. The prices for all flights offered by a specific airline are modified accordingly to an exchange rate.*

```
in { resource { "http://airline.com" },
    flights {{
      last-changes { var L replaceby var Today },
      currency { "EUR" replaceby "Dollar" },
      flight {{
        price {{ var Price replaceby var Price * var Exchange }}
       }}
     }}
   }
```

Note that the construct term `var Price * var Exchange` *of the second replace operation in the above example shares the variable* `Price` *with the query term of the replace operation. This means that for each subterm matching the price of flights a "corresponding" construct term is used for replacing.*

On Introducing Other Constructs for Specifying Insertion and Replacement in XChange (Discussion). Language constructs are introduced for easing the programming task. As already discussed in Section 2.2.3, a tradeoff between the expressive power of a language and the ease of its usage needs to be found in designing a language. Decisions need to be taken for introducing (or not) new constructs when use cases are discovered that can not be solved with the existing language constructs.

Example 2.45 (Example Motivating a New Language Construct) *Assume the XML files* `catalogue-eu.xml` *and* `catalog-usa.xml` *contain information about products a company*

provides on the European and American market, respectively. One wants to insert all products of class "New Arrival" found in the European catalogue into the American one; the product prices should be calculated and inserted directly in US Dollar.

The following query is used to select the products of class "New Arrival"; a variable is used also for the price, as this is needed later in the update term.

```
in { resource { "file:catalogue-eu.xml" },
    var Product → desc product {{
                        class { "New Arrival" },
                        price { var Price } }}
}
```

For accomplishing the task described above, an update specification is needed for inserting all bindings for Product into `catalog-usa.xml`*, where the price is calculated from the bindings for Price and the binding for a variable Exchange (this is obtained by querying another Web resource where the up-to-date exchange rates are found). Such a problem could be solved by means of a construct of the form*

```
insert Construct_Term_1
             with var VarName replaceby Construct_Term_2
```

The variable $VarName$ needs to occur as a subterm in the query term to which a variable of $Construct_Term_1$ is restricted. E.g. the variable Price is bound to a subpattern of the pattern defining Product. Moreover, $Construct_Term_1$ either contains a single variable or is of the form var $Var -> Construct_Term$, where $VarName$ occurs in the query term defining Var. Thus, the task of this example can be elegantly accomplished by evaluating the following update specification:

```
in { resource { "file:catalog-usa.xml" },
     catalog {{
        insert all var Product
                        with var Price replaceby var Price * var Exchange
     }}
}
```

Such a language construct has not been introduced in XChange, as (the class of) examples like the one given above can be realised by using the existing language constructs; however, the solution is clearly not that elegant as the one given above. (*Note* that insertions of new data terms where some subterms are left out can be easily specified by using the Xcerpt construct `except`.)

On the Keywords Chosen for Specifying Updates in XChange (Discussion). Update operations are specified in XChange by using the keywords `insert`, `delete`, and `replaceby`, considered as infinitive verbs expressing the kind of updates to perform. One might argue that these are imperative verbs giving an imperative flavour to the language. The problem of updating data has indeed an imperative nature. However, XChange is a declarative language as it specifies the *what* instead of the *how*, just like logic programming languages. Another approach is to use the participle of the verbs insert, delete, and replace by to specify the kind of changes one desires. Thus, the core XChange update operations would have the following form

```
inserted Construct_Term
deleted  Query_Term
Query_Term replacedby Construct_Term
```

As XChange builds upon Xcerpt, one of the most important issues in designing the language XChange has been the uniformity e.g. of the language constructs. Thus, as Xcerpt keywords are specified as infinitive verbs (e.g. `CONSTRUCT` and not `CONSTRUCTED`), the same approach is taken in XChange.

2.4.1.5 Special Case – Updating the Root

The insertion, deletion, and replacement specifications introduced in the previous sections augment query terms for obtaining update term specifications. The premise of modifying desired data terms is that the query terms – called subjacent query terms – simulation unify with the data terms. For *updating the root* – inserting a data term into an empty Web resource, deleting the whole data term found at a Web resource, or overwriting (replacing) the data term at a Web resource – a subjacent query term is neither needed nor allowed, a single update operation will do.

Imagine an XChange update like a function that takes as arguments an update term and a data term, and returns a (updated) data term. Data terms represent trees, thus, an XChange update can be represented as a function that applies an update term to a tree for obtaining the updated tree. Regardless whether one wants to update the root of a tree, the whole tree, or just parts of it, the result of the update needs to be a *tree*. This requirement has consequences on the possible update operations for updating the root. Their syntax is the same as for the update operations introduced so far; though, not all update specifications are allowed for updating the root so as not to violate the previously given requirement.

The following grammar rules define the possible update operations for updating the root:

```
U_Root ::= "insert"  Construct_Term
       | "delete" Query_Term
       | Query_Term "replaceby" Construct_Term
```

Insertion into an Empty Resource. In XChange, the update used for inserting data (more precisely a data term) at a given Web resource *Res* has the form

```
in { resource { Res },
     insert Construct_Term }
```

The resource specification is given to emphasize the absence of a query term acting as a subjacent one. *Note* that if the given Web resource contains data (it is not empty), the above update term constructs a new data term that overwrites the data found at *Res*. Update operations of one of the following forms

```
 insert all Construct_Term
 insert some  Nr Construct_Term  , with 0 < Nr
 insert first Nr Construct_Term  , with 0 < Nr
```

are not allowed for updating the root. They result in not having a tree representation of data, but a *forest* (a sequence of trees). If one wants to insert e.g. all instances of a construct term in an empty resource, an artificial root should be provided.

Example 2.46 (Insertion into an Empty Resource) *An update that specifies an insertion of the data term found at* `http://sn.de` *into the empty document:* `http://software.de/products.xml`*.*

```
in { resource {"http://software.de/products.xml"},
     insert var C
    }
```

The condition part of the XChange rule having the above given update term as action part contains the following Xcerpt query that binds the variable C:

```
in { resource {"http://sn.de"},
     var C → Catalogue {{ }}
    }
```

Deletion of the Root. For deleting the whole data term found at a given Web resource, a query term matching it needs to be specified. The XChange update for deleting the (data of) Web resource *Res* has the form

```
in { resource { Res },
     delete Query_Term
}
```

Clearly, if *Query_Term* has the form *desc query_pattern* the data term at *Res* remains unchanged. *Note* that an update term of the form *delete varRoot* results in deleting the data term regardless of its structure. Update terms of one of the following forms

```
delete some  Nr Query_Term  , with 0 < Nr
delete first Nr Query_Term  , with 0 < Nr
```

are not allowed. They do not have an intuitive meaning when updating the root; thus, are not considered as update terms in XChange.

Example 2.47 (Deletion of the Root) *An update that specifies the deletion of the data term found at* `http://software.de/products.xml` *(i.e., the entire XML document is to be deleted).*

```
in { resource {"http://sn.de"},
    delete Catalogue {{ }}
}
```

Replacement of the Root. For replacing the data term found at a Web resource with a new data term, XChange offers the following update specification:

```
in { resource { Res },
     Query_Term replaceby Construct_Term }
```

Update terms specifying multiplicity are not allowed, neither for *Query_Term*, nor for *Construct_Term*. The same motivation as for insertion and deletion. As for insertion, if the resource *Res* contains data, by applying the following update the data constructed with *Construct_Term* is to be found at *Res* (overwriting effect):

```
in { resource { Res },
     var Root replaceby Construct_Term }
```

Example 2.48 (Replacement of the Root) *An update that specifies the replace of the data term found at* `http://sn.de` *with the data term found at* `http://software.de`*.*

```
in { resource {"sn.de"},
    Catalogue {{ }} replaceby var P
}

in { resource {"http://software.de"},
    var P → Products {{ }}
}
```

On Alternative Approaches to Updating Data with XChange (Discussion). In contrast to existing proposals for update languages for the Web, XChange is a pattern-based language, which shows that *update patterns* (called update terms in XChange) can be easily specified and understood. XChange takes the philosophy of Xcerpt and applies it to specifying evolution of data on the Web; Xcerpt uses the pattern approach for querying data on the Web – capability needed (as already shown in the previous sections) for updating (persistent) data and for querying (volatile) data. For updating Web data with XChange, alternative approaches have been investigated; this discussion concentrates on these approaches by showing the evolution towards the current update component of XChange.

Following the first approach that has been investigated, the desired updates have been executed directly on bindings for variables returned from evaluating an Xcerpt query. Xcerpt queries are specified as patterns for the data and do not select nodes, but return data terms as bindings for the variables in the query – they are *copies* of data terms. Thus, performing updates *directly on* the data terms obtained as answers to Xcerpt queries means that only copies of the data are modified; such updates *do not update* (modify) the data. Recall Example 2.44, which specifies the modification of the used currency from Euro to US Dollar. Consider now the following query term that is needed for binding data terms to variables and modify them afterwards:

```
in { resource { "http://airline.com" },
    flights {{
      var Ch ->last-changes { var L },
      var Curr -> currency { "EUR" },
      flight {{
        var P -> price {{ var Price }}
       }}
     }}
   }
```

The modification of prices can then be realised by using the following update specification (one needs to specify where the subterms to be modified are – through the *in Variable* specification, a query term and a construct term for the replace operation):

```
in { resource { "http://airline.com" },
     in var Ch replace var L by var Today,
```

```
    in var Curr replace "EUR" by "Dollar",
    in var P replace var Price by var Price * var Exchange
  }
```

A solution to the problem of updating only copies of data when using pattern-based query languages like Xcerpt is to enhance queries with the capability to return pointers to elements inside data terms. The pointers to data are then used in update specifications. On the other hand, by using such an approach, patterns and pointers for data would be mixed. For having a clear language design, a single approach has been followed in XChange – the pattern-based one – leading to an elegant language, easy to understand and use by practitioners.

Another idea for updating persistent data is to give the intended updates "implicitly" by specifying how data should look after the updates are performed; following such an approach consists in specifying the *result* of the updates instead of the *way* (given through explicit update operations) towards the desired result. This would mean that, instead of XChange update terms, construct terms are used that give a pattern for the data after the update. For determining the update operations to be executed, a diff function can be used. This idea has not been incorporated into XChange, as specifying the result of the updates is not that easy as it seemed at first glance. A small set of update operations and update constructs gives rise to simpler update specifications. However, thinking in the other direction – to translate XChange update terms into deductive, construct-query rules – has yielded interesting results that are given in Section 2.4.3.

2.4.2 Complex Updates as Transactions

XChange updates may be *elementary* or *complex*. An *elementary update* is a change (i.e. insert, delete, replace) to a persistent data item (e.g. XML or RDF data) that can be expressed by means of an update term. *Complex updates* expressing ordered or unordered conjunctions, or disjunctions of (elementary or complex) updates are also offered by XChange. Such updates are often required by real applications. E.g. when booking a trip on the Web, one might wish to book an early flight *and* the corresponding hotel reservation, *or* else a late flight *and* a shorter hotel reservation. Since it is sometimes necessary to execute such complex updates in an *all-or-nothing manner* (e.g. when booking a trip, a hotel reservation without a flight reservation is useless), XChange has a concept of transactions.

The grammar rule defining XChange updates is the following (an XChange update is an elementary or a complex update):

```
Update     ::= E_Update | C_Update
```

The next section discusses elementary updates by shortly revisiting the notion of update patterns introduced in the previous section. Complex updates are combinations of elementary and complex updates; they are introduced in Section 2.4.2.2. XChange transactions, i.e. XChange updates executed in an all-or-nothing manner, are discussed in Section 2.4.2.3.

2.4.2.1 Elementary Updates

An *elementary update* specification is an update term specification accompanied by a resource specification (giving the Web resources to be modified). Section 2.4.1 offered an introduction to XChange update terms – patterns for the data to be updated augmented with the desired update operations. The three kinds of XChange update operations (insertions, deletions, and

replacements) have been described almost independent from each other (Sections 2.4.1.2, 2.4.1.3, and 2.4.1.4). However, more than one update operation can be specified in an update term (nesting update operations does not make sense). Update operations are specified as subterms inside a query term – called subjacent query term of the update term. Though, for updating the root of a data term (e.g. inserting data into an empty resource or deleting the whole data term found at a given Web resource), no subjacent query term is needed – a single update operation is used.

Definition 2.3 (Subjacent Query Term of an Update Term) *Given an update term u, the subjacent query term of u (denoted sub_u) is an Xcerpt query term obtained by removing from u*

(a) the insertion operations, except the query terms they may contain due to `after` *or* `before` *position specifications;*

(b) the `delete` *keywords and the multiplicity specifications of delete operations;*

(c) the replace operations, except the query terms they contain.

Steps (a), (b), and (c) can be performed in arbitrary order for obtaining sub_u.

For example, for insertion specifications of the form $after\ Query_Term\ insert\ Construct_Term$, the $after$ and $insert\ Construct_Term$ are to be removed for obtaining the subjacent query term of an update term containing the update operation. Subjacent query terms of XChange update terms play an important role in the execution of the desired updates.

Proposition 2.2 (Role of Subjacent Query Terms) *Let $U = (d, u)$ be an elementary update with data term d to be modified by update term u. If $sub_u \preceq d$ does not hold, no update operations of u are to be applied to d.*

If the subjacent query term of an elementary update matches the data term to be modified, the specified update operations are executed. XChange update operations are *intensional updates*, they are a description of updates in terms of (standard or event) queries. They can be specified in XChange as the language inherits the querying capabilities of the language Xcerpt.

Example 2.49 (Elementary Update) *At* `http://airline.com` *the flight timetable needs to be updated as reaction to flight cancellations. The information about the cancelled flight is obtained from the event part of the rule having the following elementary update as action part.*

```
in { resource { "http://airline.com" },
    flights {{
      last-changes { var L replaceby var RTime },
      flight {{ number { var N }, date { var RTime },
                delete departure-time {{ }},
                delete arrival-time {{ }},
                insert news { "Flight has been cancelled!!" }
       }}
     }}
   }
```

2.4.2.2 Complex Updates

An XChange *complex update* is an ordered or unordered, conjunction or disjunction of updates (i.e. of elementary or complex updates). A conjunction of updates expresses that all specified

updates are to be executed. A disjunction of updates expresses that one of the specified updates is to be executed.

Complex updates specifying conjunctions are introduced by the keyword `and`, disjunctions by the keyword `or`. Specifications denoting that the XChange update is a complex one are always total (i.e. partial conjunctions or disjunctions of updates do not make sense). Square brackets and curly braces are used for denoting that the order of evaluation is of importance or of no importance, respectively. The grammar rules defining complex updates in XChange are given next:

```
C_Update     ::= Ordered_CU | Unordered_CU
Ordered_CU   ::= "and" "[" Update_List "]"
             | "or" "[" Update_List "]"
Unordered_CU ::=  "and" "{" Update_List "}"
             | "or" "{" Update_List "}"
Update_List  ::= Update ("," Update)+
```

XChange offers four kinds of complex updates – ordered and unordered conjunctions, ordered and unordered disjunctions of updates. The effect of such updates and the scope of variables occurring in complex updates are explained in the following for each of these kinds.

Ordered Conjunction of Updates. Consider an XChange complex update specification o_conj of the form $o_conj = and[u_1, u_2, ..., u_n]$, where $2 \leq n$ and u_i specify XChange updates. The *effect* of u_conj is the effect of executing all u_i ($1 \leq i \leq n$) sequentially in the order of their occurrence in the list. This means that the effect of an update u_i is “visible” for updates u_j, with $j > i$. The visibility of update effects is twofold:

(a) the bindings for the variables of u_i that are obtained by evaluating u_i can be used in the evaluation of u_j with $j \leq i + 1$;

(b) consider updates u_i and u_j with $j > i$ that modify the data found at a resource Res, and there is no u_k with $i < k < j$ that modifies Res. For a data term d at Res before the updates are performed, u_i modifies d and results in having at Res a data term d_i, whereas u_j modifies the data term d_i.

Example 2.50 (XChange Complex Update Specifying Sequence of Updates) *The following XChange complex update specifies that a flight reservation and a hotel reservation are to be executed in the specified order. After giving the shape of such an update, an instantiation of it follows.*

```
and [
  <make flight reservation>,
  <make hotel reservation depending on the flight schedule>
  ]
```

The following complex update specifies that a flight and a corresponding hotel reservation are to be made for Christina Smith. The bindings for the variables F (the chosen flight) and H (the chosen hotel) are obtained from the other parts of the rule having the update as action part. Note that the variables N, B, and E are bound during the evaluation of the first update and used afterwards in evaluating the second update of the conjunction.

```
and [
   in { resource { "http://travel-agency.net/flights/" },
        desc reservations {{
          insert reservation {
                  var F, var N -> name { "Christina Smith" },
                  outward-date { var B ->"2005-08-21" },
                  return-date { var E ->"2005-08-22" } }
        }}
      },
   in { resource { "http://hotels.net/reservations/" },
        accommodation {{
          insert reservation {
                  var H, var N,
                  from { var B }, until { var E }  }
        }}
       }
  ]
```

As the example above shows, ordered conjunction of updates is amenable to applications involving sequences of updates to be executed, where the order of update execution plays an important role. They are useful when data gathered or used in an update is needed for executing subsequent ones, or when complex modifications to the same Web resources' data are needed.

Unordered Conjunction of Updates. Consider an XChange complex update specification u_conj of the form $u_conj = and\{u_1, u_2, ..., u_n\}$, where $2 \leq n$ and u_i specify XChange updates. The *effect* of u_conj is the effect of executing all u_i, $1 \leq i \leq n$, in some arbitrary execution order. The order of their execution is not given and, thus, the runtime system has the freedom to choose the execution order.

The *scope* of variables used in update u_i is restricted to u_i, i.e. the bindings for the variables resulted from evaluating u_i can not be used in the evaluation of u_j with $i \neq j$. (This restriction can be lifted, for parallel evaluation of updates; variable substitutions need to be communicated between the Web sites where the data to be modified are found.) Unordered conjunction of updates are suitable for specifying updates to be executed that do not "share" other variable bindings than the ones received from the event query and/or Web query of the rules whose action they represent.

Note that unordered conjunction of updates that modify the same data may have different results, depending on the order of their evaluation. This is illustrated by means of an example.

Example 2.51 (Complex Update Specifying Unordered Conjunction of Updates) *The following example specifies that a deletion and an insertion should be executed on the same data, in* `text.xml`.

```
and {
  in { resource { "file:test.xml" },
       a {{ delete b {{}} }},
  },
  in { resource { "file:test.xml" },
```

```
      a {{ insert b { f { "content"} } }},
  }
}
```

Consider that `test.xml` *contains the following data term before any of the updates specified above are executed:*

```
a {
  b { f { "info" } },
  b { g { "info" } },
  c { h { "info" } }
}
```

The possible results of executing the unordered conjunction of updates on `test.xml` *are given next. Depending on the order in which the two updates are executed, one of the following data terms are obtained (on the left hand side the result of executing the deletion followed by the insertion, on the right hand side the result of insertion followed by the deletion):*

```
a {
  b { f { "content" } },
  c { h { "info" } }
}
   Delete, insert
```

```
a {
  c { h { "info" } }
}
Insert, delete
```

Ordered Disjunction of Updates. Consider an XChange complex update specification o_disj of the form $o_disj = or[u_1, u_2, ..., u_n]$, where $2 \leq n$ and u_i specify XChange updates. The *effect* of o_disj is the effect of executing one single u_i, $1 \leq i \leq n$; the disjunction of updates is an exclusive one. The ordered specification expresses that the runtime system should try to execute the updates in the given order, until a (first) update has been successfully executed. Like for unordered conjunctions of updates, the *scope* of variables used in update u_i is restricted to u_i.

Example 2.52 (XChange Complex Update Specifying Disjunction of Updates) *The following XChange complex update specifies that a travel reservation is to be performed, if no flight reservation can be made, a train ticket should be reserved. The disjunction update has the following shape:*

```
or [
  <make flight reservation>,
  <reserve train ticket>
  ]
```

The above template is instantiated to make the desired reservation for Christina Smith; bindings for the variables are obtained from evaluating the event query and Web query of the rule having the update as action. Note that one cannot bind a variable to name{"Christina Smith"} and then use it in the other update of the disjunction.

```
or [
  in { resource { "http://lhs.de/flights/" },
```

```
      desc reservations {{
        insert reservation {
                var Flight, name { "Christina Smith" }  }
      }}
    },
in { resource { "http://db.de/trains/" },
      desc tickets {{
        insert reservation {
                var Train,  name { "Christina Smith" }  }
      }}
     }
]
```

Unordered Disjunction of Updates. Consider an XChange complex update specification u_disj of the form $u_disj = or\{u_1, u_2, ..., u_n\}$, where $2 \leq n$ and u_i specify XChange updates. The *effect* of u_disj is the effect of executing one single u_i, $1 \leq i \leq n$. The runtime system has the freedom to choose the order in which it tries to find and successfully execute one of the updates. Like for ordered disjunction of updates, the *scope* of variables used in update u_i is restricted to u_i.

2.4.2.3 Transactions

An XChange *transaction specification* is a group of (elementary or complex) update specifications and/or explicit event specifications (expressing events that are constructed, raised, and sent as event messages) that are to be executed in an *all-or-nothing manner*. That is, either all specified actions are successfully executed or none of the updates is executed (partial effects of the updates need to be undone).

Elementary and complex update specifications have been introduced in the previous sections. They specify (local or remote) Web resources to be modified and the updates to be performed on their data. An XChange *event specification* is a (complete) *pattern* for the event message(s) to be constructed and sent to one or more Web sites. The notion of *event terms* is used to denote such patterns for events to be raised. An event term represents a restricted construct term that may be preceded by the keyword `all`.

A restricted construct term is an Xcerpt construct term having root labelled `XChange-Namespace:event` and at least one subterm `XChange-Namespace:recipient`$\{uri\}$ that specifies a Web site's address. The constructed event message is to be sent to the XChange program found at *uri*. If more than one subterms of the form `XChange-Namespace:recipient` $\{uri\}$ are given in an event term, the constructed event message is to be sent to all specified recipient Web sites.

An event term of the form `all Construct Term` is used to raise and send *all* events that are constructed with `Construct Term` by applying the substitutions obtained from the rest of the XChange reactive rule whose head specifies the event term. Such event terms are useful e.g. when the event messages to be sent have different content (depending on the variable substitutions).

Actually, $Event_Term$ in the grammar rules given next is not a construct term with arbitrary structure – it has been generalised to construct term for reasons of simplicity. The following grammar rules define transactions specifications in XChange:

```
Trans      ::= Update
           | Event_Term
           | Ordered_AList
           | Unordered_AList
Event_Term      ::= Construct_Term
                | "all" Construct_Term
Ordered_AList   ::= "and" "[" AList "]"
                | "or" "[" AList "]"
Unordered_AList ::=  "and" "{" AList "}"
                | "or" "{" AList "}"
AList  ::= Update ("," Action)+
Action ::= Update | Event_Term
```

A transaction specification can be considered as an ordered or unordered conjunction or disjunction of *action* specifications. Currently, updates and event terms are considered as actions in XChange. However, this view offers flexibility in extending XChange with other kinds of actions if considered necessary. The discussion on visibility of update effects for other updates inside a transaction (including also the usage of obtained variable substitutions) can be ported to the more general setting of actions (and covering thus event terms or combinations of updates and event terms).

On XChange Transactions and their Management on the Web. Combinations of XChange actions are considered transactions if they obey the ACID properties [71] (Atomicity, Consistency, Isolation, and Durability). Communicating transaction requests and synchronising the actions to be taken can be implemented to some extent by means of XChange rules; however, transaction-related issues deserve more investigation in the framework of XChange so as to realise transaction management on the Web. The idea is to extend XChange with standard solutions from database systems that are to be adapted to the biggest existing distributed system – the Web.

Actions performed inside of a distributed transaction on the Web may trigger local or remote actions that in turn can trigger other actions (i.e. *cascading triggering* on the Web). Upon abort of a triggering transaction, rollback of all triggered actions needs to be assured on the Web, a decentralised environment posing new challenges. A nested transaction model has been proposed in [74] (for HiPac, pages 184 - 186) for accommodating with the relationship between a transaction and the rules triggered by it (which in turn can trigger other rules). How to cope with these kinds of problems has been discussed also in [74] (for Chimera, page 167) and [60]. Existing proposals for management of triggering transactions and triggered ones in database systems might prove very useful in extending XChange.

Transactions defined at the level of an XChange-aware Web site, or more concrete in an XChange program, should recognise contradictory transactions and update specifications, possibly at compile time, or develop transaction inconsistency resolution strategies. For example, for *insert a* and *delete a*, conceivable strategies would be to execute the update associated with the reactive rule with higher priority, or not to execute these two updates at all. At moment, however, XChange does not consider priorities for rules but, at the same time, leaves room for such kind of extensions.

Contingency mechanisms could be also employed for transaction management on the Web, i.e. use the events expressing abort of a transaction to specify how to react in case that a

transaction aborts.

The emphasis in this report is not on a language for distributed transactions on the Web; we recognise the need for transactions through developed application scenarios and the components a transaction on the Web might have, and proposes a syntax for event-driven transactions. A complete investigation and realisation (including formal semantics and implementation) of transactions on the Web are outside the scope of this report, and are to be further developed later in the project.

2.4.3 Semantics of Updates

This section discusses the declarative semantics of elementary and complex updates in XChange. Recall that transactions – combining updates and events to be raised – can be specified in XChange. Though, this work on declarative semantics is restricted to XChange updates, as the accent in this report is not on a language for distributed transactions on the Web; we recognise the need for transactions through developed application scenarios and the components a transaction on the Web might have, and proposes a syntax for event-driven transactions. The declarative and operational semantics of transactions on the Web are outside the scope of this report.

Semantics of Elementary Updates. An XChange elementary update specification consists of a resource specification and an update term: The resource specification gives the location and names of the documents to be updated. The update term is a pattern for the data to be updated augmented with the desired update operations. The effect of an elementary update is that the data at the given resources has been 'refreshed' according to the given update term. The same result can be obtained when *constructing* the data after the update. In the context of XChange this means that an elementary update has the same effect as an Xcerpt *goal* that constructs the data after the update.

Given an elementary update u for modifying resources Res_i, $1 \leq i \leq n$ (a finite number of Web resources) with update term t^u, a corresponding Xcerpt goal G_u exists that constructs new data at Res_i. This (new) data constructed at Res_i is the data that would be obtained by applying t^u on the (old) data at Res_i. *Note* that data constructed by G_u overwrites the old data found at the resources. A set of rewriting rules have been recognised that rewrite an elementary update u into an Xcerpt goal G_u such that the effect of G_u is the same as the effect of u. The underlying ideas of rewriting are given in Section 2.4.4.

Thus, the semantics of an elementary update u can be reduced to the semantics of a corresponding Xcerpt goal G_u of the form $t^c_u \leftarrow \mathcal{Q}_\sqcap$. That is, the model theoretical semantics of Xcerpt, whose underlying ideas have been presented in the previous section, can be used for defining the semantics of elementary updates in XChange.

An elementary update u is transformed into a goal $t^c_u \leftarrow \mathcal{Q}_u$ and gets, thus, a formula representation $\forall^* \ll t^c_u \leftarrow \mathcal{Q}_u \gg \wedge \bigwedge_{1 \leq i \leq n} d_{Res_i}$, where d_{Res_i} are the data terms to be modified. The satisfaction of such a formula in an interpretation is defined in Definition 2.1. Intuitively, the model for the formula expressing the update contains the data terms after the update has been performed.

Example 2.53 (Declarative Semantics of Elementary Updates) *Consider the following update term:*

```
bib {{
      book {{ price { var P replaceby var P * 1.5 } }}
}}
```

Assume that the data term to be updated contains books listed with their titles and prices; each book price needs to be modified. A goal corresponding to the above given update term has the following formula representation:

```
∀ P ≪bib { book { price { var P*1.5 },
all var O }, all var C } ← bib {{ book {{ price
{ var P }, var O }}, var C }} ≫
∧ bib { currency {"Euro"}, book { title {"Linux in a
Nutshell"}, price {"36"} }, book { title {"Data on the Web"},
price {"40"} } }
```

A model for the above given formula is $M = (I, \Sigma)$ *where:*

```
I = {
   bib { currency {"Euro"}, book { title {"Linux in a Nutshell"}, price {"54"} },
                            book { title {"Data on the Web"}, price {"60"} } },
   bib { currency {"Euro"}, book { title {"Linux in a Nutshell"}, price {"36"} },
                            book { title {"Data on the Web"}, price {"40"} } }
} Σ = { {∅} }
```

The interpretation I *contains the data terms before and after the update. The formula corresponding to the update term together with the data term to be modified represents an Xcerpt program that produces the data term where the book prices are replaced by the new prices.*

Semantics of Complex Updates. Complex updates specify conjunctions or disjunctions of (elementary or complex) updates. Based on the formula representation of elementary updates and using the connectors $\wedge$ and $\vee$ the formula representation of complex updates are constructed. A formula of the form

(i) $F_1 \wedge \cdots \wedge F_n$ represents a complex update specifying conjunction of updates,

(ii) $F_1 \vee \cdots \vee F_n$ represents a complex update specifying disjunctions of updates,

where F_i is the formula representation of an elementary or complex update. As for elementary updates, the Definition 2.1 is used for determining if such a formula is satisfied in an interpretation or not.

Remark. Observing that the effect of an elementary update can be 'simulated' by an Xcerpt goal, the model-theoretical semantics of the query language Xcerpt has been used for defining the declarative semantics of update language of XChange. However, this approach does not cover ordered conjunctions and disjunctions of (elementary or complex) updates. Ordered complex updates enforce an order for performing the given updates. Moreover, some specified updates might depend on updates that are to be executed before them (so as to use bindings for the variables obtained after update execution). These features cannot be defined by means of a model-theoretical semantics.

2.4.4 Execution of Updates

This section discusses the execution of XChange updates. It considers first some conceivable approaches for executing updates on Web data (updates specified by means of an update language). Then it recalls the model for updating data on the Web and the update operations considered in this work. The section ends by presenting the approach taken in XChange and already mentioned in Section 2.4.3; the underlying ideas of the taken approach are given and explained in detailed steps through a simple example of an XChange update.

Executing Updates to Web Data. Different approaches are conceivable for executing the update operations specified by means of an update language for Web data; they are determined by different conditions or criteria that need to be taken into account. First, updating data depends on the representation formalism and the storage of the data to be modified. One can find data on the Web (as Web resources data) represented in a multitude of formalisms (such as HTML, XML, RDF, or relational databases) and data storage (e.g. XML data can be stored as native XML documents, in relational databases such as Tamino [68], or in object-oriented databases). Clearly, the representation of data is more important for the language constructs and the storage of data for the execution of updates specified through these constructs.
Let's now consider another 'dimension' of update execution. Updates can be performed

(i) *in-place*, meaning that the specified update operations are executed directly (data need not loaded in the memory) on the data to be updated, or

(ii) *in-memory*, meaning that the data to be updated is loaded in the memory where it is modified, the data after the update need to be 'placed instead' of the old (initial) data. Using this approach, the data after the update can be constructed in memory or the update operations can be performed on the internal representation of data (e.g. on the DOM representation when updating XML data).

Also, an update language can have proprietary update execution abilities, or transformation rules can be provided for the language constructs into existing 'update management' means. For the latter case, a mapping between the update constructs of the language and update or construction constructs of another language are provided; evaluating or executing the obtained programs yields the same effect as if a proprietary update processor is available. For example, update operations on XML documents can be mapped into SQL update operations that work on an XML (relational) database; such XML-to-SQL-mappings eliminate the need to understand the database structure.

Efficiency issues can play an important role when updating data on the Web; however, there are few proposals for efficient execution of updates on Web data having (native) XML storage. For example, finding the least expensive sequence of operations to transform an initial document (before any update is performed) in the final one (the document after the specified update are performed) pose interesting research problems.

XChange Updates on the Web. An XChange program is located at one (XChange-aware) Web site and contains rules specifying ordered or unordered conjunctions and/or disjunctions of updates. Updates are specified as "action part" of XChange reactive rules; they are performed after the successful evaluations of the other parts (event query, Web query) of the rules. Thus, the substitution set $\Sigma_u = \Sigma_{eq} \cap \Sigma_{wq}$ is used for performing the specified updates, where Σ_{eq}

and Σ_{wq} are the substitution sets obtained from evaluating the event query part and the Web query part, respectively.

XChange updates express how data found at one or more Web resources is to be modified, i.e. how persistent data is to modified. These Web resources are either local or remote. Updates to local Web resources are executed by the language processor at the Web site. Updates to remote Web resources are *not* executed by the processor of the Web site where the update has been specified; instead updates to remote data are *update requests* to the Web sites where the data to be modified is stored. A Web site receiving an update request can try to execute the update or decide not to execute the requested update. This approach is consistent with the local control of XChange programs.

An XChange elementary update consists of a resource specification (the resources to be updated) and an update term (a pattern for the data to be modified augmented with update operations). The subjacent query term of an update term is the underlying query pattern of the respective update term. Consider an elementary update u specifying modifications of data term d_u through an update term t_u whose subjacent query term is s_{t_u}; a premise for a successful execution of the update operations of u is the satisfaction of the condition $s_{t_u} \preceq d_u$. In other words, the query s_{t_u} needs to evaluate successfully against d_u for performing the given update operations on d_u. The evaluation of the subjacent query term of an update term against the given data to be modified can represent a 'pre-update operations execution' step for determining whether to (try to) execute the update operations on these data or not. However, performance results need to be compared for executing updates with and without subjacent query term evaluation for determining which technique is more suitable and less expensive.

XChange update operations specify insertions, deletions, or replacements of data for tree-like Web data. Executing an update operation

- `insert ConstructTerm` implies the construction of a data term using `ConstructTerm` and the variable substitutions obtained from the other parts of the rule and the evaluation of subjacent query term; the construction follows closely that of Xcerpt [64]. Where the constructed data term is inserted is given by the position of the insertion operation inside the subjacent query term.

- `delete QueryTerm` deletes all terms matching the `QueryTerm`; all subterms of these terms are deleted.

- `QueryTerm replaceby ConstructTerm` replaces all terms matching the `QueryTerm` with a data term constructed with `ConstructTerm`.

The other constructs for XChange update operations are executed conforming to their meaning (that have been introduced informally in Section 2.4.1), for example by inserting at a given position in the document when a position is specified.

An XChange complex update specifies ordered or unordered conjunctions or disjunctions of (elementary or complex) updates. Executing a complex update

- `and` $[U_1, \ldots, U_n]$ means executing all U_i, $1 \leq i \leq n$ in the given order so as to use a substitution set for the variables obtained by executing U_i in the subsequent updates U_j, $i+1 \leq j \leq n$.

- `and` $\{U_1, \ldots, U_n\}$ means executing all U_i, $1 \leq i \leq n$ regardless of the execution order (unordered updates can be executed in parallel).

- `or` $[U_1, \ldots, U_n]$ means executing one of the U_i, $1 \leq i \leq n$ by trying to execute the updates in the given order and stop after the first successful execution of a specified update.
- `or` $\{U_1, \ldots, U_n\}$ means executing one of the U_i, $1 \leq i \leq n$; the processor can pick freely the update to be executed from the given ones.

The execution of complex updates is a kind of controlled execution of two or more elementary or complex updates; the building block consists in executing XChange elementary updates. The approach taken in XChange for executing elementary updates is discussed in the following.

Updates through Construction. The approach taken for executing XChange updates is an in-memory execution of updates where the elementary updates are executed by constructing the data after the update. Mappings between XChange elementary updates and Xcerpt goals are provided. Data to be updated has a tree-like representation (e.g. XML, or RDF data) stored as XML documents.

XChange updates specifying modifications as an alternative to an intentional specification, i.e. constructing the data after the update by means of deductive rules. That is, for each XChange elementary update a corresponding Xcerpt goal exists, such that evaluating the Xcerpt goal has the same effect as if the respective update operations were executed directly on the data. The substitution set obtained from evaluating the other parts of the rule having the elementary update as action or from executing other specified updates in the action part is used in evaluating the Xcerpt goal. The data term constructed by evaluating the Xcerpt goal is the data term after the update; this modified data term overwrites the initial data (the data before the update).

Given an XChange elementary update u, a corresponding Xcerpt goal G of the form $ConstructTerm \rightarrow_g QueryTerm$ is constructed by taking the structure of the subjacent query term and the update operations of u into account. This transformation poses the following challenges:

(i) *partial patterns* in u do not offer knowledge about all subterms of terms in the data to be modified; means are needed for determining whether a term has other subterms than those specified in the query pattern, and for gathering all these subterms if they exist so as not to loose data through construction.

(ii) the *original order* of the terms in the data to be modified needs to be maintained.

(iii) the *semantics* of XChange update operations needs to be mirrored by the constructed Xcerpt goals.

(iv) the *position specifications* in insertion operations express insertion of data at the given position; means are needed so as to assure that the modified data contains the inserted data at the given position.

Rewriting rules have been developed for rewriting an XChange elementary update specification into a corresponding Xcerpt goal specification. These rules are applied recursively on the structure of the given update term to obtain a tuple $(ConstructTerm, QueryTerm)$ consisting of the construct and query part of an Xcerpt goal. The resources given in the elementary update are 'forwarded' to the query and construct part of the Xcerpt goal. The rewriting rules comprise the following solutions to the problems mentioned above:

(i) partial patterns imply the use of Xcerpt's construct `optional` before a fresh variable in the goal's query and gathering of all these in its construct term (so as to ensure that no data is to be loosed along the construction way);

(ii), (iv) the desired order of the subterms in the modified data term is assured by combining the use of ordered patterns with Xcerpt's ordering of terms based on their position.

(iii) the desired effect of XChange update operations is achieved by a careful development of the rewriting rules based on the fact that XChange update terms consist of Xcerpt terms (query terms and construct terms) and update constructs.

The following example explains the transformation steps that are needed in order to go from an XChange elementary update to a corresponding Xcerpt goal.

Example 2.54 (Flight Reservation Specified as Deductive Rule) *Recall Example 2.56 specifying an XChange transaction rule for booking another flight as reaction upon a flight cancellation. After evaluating the event query and Web query parts, the specified action is to be performed. Its specification follows:*

```
in { resource { "http://airline.com/reservations/" },
     reservations {{
        insert reservation { var F, name { "Christina Smith" } }
       }}
    }
```

For constructing a corresponding Xcerpt goal for the above given XChange elementary update simple steps need to be taken by paying attention to the structure of the update term. The resource `http://airline.com/reservations/` *to be modified is treated in a straightforward manner: the query part of the goal queries it and the construct part of the goal specifies it as the output resource (where the data after the update should be 'put').*
The subjacent query term of the given update term is

```
reservations {{  }}
```

which is transformed by applying the rewriting rules into a query term to be used in the query part of the goal and a construct term to be used in the construct part of the goal. The query term is obtained by adding a pattern that matches with the subterms of the `reservations` *term; the subterm and its position are to be bound to the variables* `Child` *and* `CPos`*, respectively. The* `optional` *construct is used because there is no knowledge about the existence of other made reservations (or subterms of other kind) at* `http://airline.com/reservations/`*. Thus, the following query term is obtained*

```
reservations {{
   optional position var CPos  var Child
}}
```

Let's turn attention to the construction of the goal's construct term. The partial specification turns into total ordered specification so as to keep the order of the subterms as in the initial data term (i.e. the data term before the update is performed). All subterms of the root `reservations` *in the initial data term are gathered by means of the construct* `all` *(so as not to loose information) and ordered by their position in the initial data term (for keeping the initial order). After these steps, the construct term looks like*

```
reservations [
    all optional var Child order by [ var CPos ]
]
```

However, using this construct term one just constructs the initial data term without taking the insertion update into consideration. One more step needs to be made; the construct term specified in the insertion operation (i.e. after the `insert` *keyword) is a pattern used in the goal's construct term for constructing a new subterm of* `reservations`. *Thus, the goal's construct term is complete as:*

```
reservations [
    all optional var Child order by [ var CPos ],
    reservation { var F, name { "Christina Smith" } }
]
```

The corresponding Xcerpt goal built by making the described transformation steps to the elementary update given at the beginning of this example is the following:

```
GOAL
  out { resource { "http://airline.com/reservations/" },
        reservations [
             all optional var Child order by [ var CPos ],
             reservation { var F, name { "Christina Smith" } }
        ]
      }
FROM
  in { resource { "http://airline.com/reservations/" },
       reservations {{
            optional position var CPos  var Child
        }}
     }
END
```

The desired booking is realised (i.e. the insertion update is executed) by evaluating the above given Xcerpt goal. The reservation for Christina Smith is to be found as the last subterm of the `reservations` *term in the data at* `http://airline.com/reservations/`.

The previously given example of an XChange elementary update is a simple one that keeps its simplicity in the transformation process and the corresponding Xcerpt goal specification. However, more complex XChange updates are still easy to be transformed (the rewriting rules are applied recursively on the structure of the update terms) but lack clear, simple specifications of the corresponding goal for programmers. As the whole transformation remains hidden and can be realised by automatic means, programmers need just to use the elegant XChange update operations.

2.5 XChange Rules

2.5.1 Rules

An XChange program is located at one Web site and consists of one or more rules: Reactive rules of the form *Event query – Web query – Action* specify situations of interest and the actions

to be automatically executed if such situations occur. Deductive rules are Xcerpt rules that infer new data from existing (persistent) Web data (are views over persistent data).

The language XChange has been deliberately designed to mirror the clear separation between *volatile* and *persistent* data. The language constructs deal either with volatile or with persistent data for easing their understanding and usage. There are two levels of the language that mirror the view over the Web data: (a) rules' level, and (b) reactive rule components' level.

(a) XChange reactive rules specify reactions to be executed in response to incoming volatile data. In contrast, XChange deductive rules deal only with persistent data, they query persistent data and construct new persistent data. XChange deductive rules are rules expressed in the query language Xcerpt (integrated into XChange), which has been described in [64], and further developed in Rewerse WG-I4, and are not the focus of this report.

(b) Regarding the components of XChange reactive rules, the *Event query* refers to (queries) volatile data and the *Web query* refers to (queries) persistent data. The *Action* might refer to volatile data (by sending event data) or to persistent data (by updating persistent data). Rule components communicate only through variable substitutions. Substitutions obtained by evaluating the event query can be used in the Web query and the action part, those obtained by evaluating the Web query can be used in the action part.

There are two kinds of XChange reactive rules that differ in the action to be executed: *Event-raising rules* specify explicit events to be constructed and sent to one or more Web sites. *Transaction rules* specify transactions to be executed. Thus, the grammar rules defining XChange rules are the following:

```
XCRule     ::= React_Rule | Xcerpt_Rule
React_Rule ::= Raise_Rule | Trans_rule
```

XChange rules can be defined by programmers, system administrators, and non-programmers with a level of knowledge depending on their application requirements. Being a high-level language, XChange should not be too difficult to use even by non-experienced programmers. Moreover, a visual counterpart of XChange is planned that could increase the accessibility of the language. XChange rules can be defined also by applications – e.g. rules can be automatically generated based on the dependencies between Web resources' data.

This section is structured as follows: Section 2.5.2 discusses XChange event-raising rules. Section 2.5.3 focuses on XChange transaction rules. Deductive rules in XChange are motivated through an example in Section 2.5.4. This section ends by defining the range restriction of XChange rules (Section 2.5.5).

2.5.2 Event-Raising Rules

XChange *event-raising rules* are means for notifying reactive (XChange-aware) Web sites of (atomic or composite) events that have occurred. They specify event messages to be constructed and sent to other Web sites as reaction on (local or remote) events. Conditions that need to hold for raising events can be specified by means of queries to persistent data; these conditions select data items from persistent data that are used for constructing event messages.

Event-raising rules are introduced by the keyword `RAISE` followed by an event term, the (atomic or composite) event query is preceded by the keyword `ON`, and the Web query by the keyword `FROM`. Event term specifications have been introduced in Section 2.4.2.3. Event queries have been discussed in Section 2.2.3. Web queries are Xcerpt queries [64]. The grammar rule defining event-raising rules is given next. (*Note* that just the event term specification is

mandatory in XChange event-raising rules; the event query and/or the Web query can be left out.)

```
Raise_Rule ::= "RAISE" Event_Term ("ON" EvQ)? ("FROM" Query)? "END"
```

Incoming events are queried by means of event query EvQ. For each answer to EvQ, the Xcerpt query $Query$ is evaluated. If $Query$ evaluates successfully, an event message is constructed from $Event_Term$ and is sent to the specified recipient. EvQ and $Query$ select data from incoming events (volatile data) and Web resources (persistent data), respectively, as bindings for the variables occurring in the queries. Assuming that the answers to EvQ and $Query$ contain the substitution sets Σ_{EvQ} and Σ_{Query}, respectively, for constructing the event message substitution set $\Sigma = \Sigma_{EvQ} \bowtie \Sigma_{Query}$ will be used.

An event that has been raised at a Web site (i.e. its representation has been constructed as event message to be sent to one or more Web sites) contains as parameters the *recipient* Web site (that needs to be given in the event term specification), the *sender* Web site, and the *raising time* (the last two are determined by the event manager of the Web site sending the event message; the event manager provides this information by inserting it into the event representation before its sending). The *reception time* and the event *id* parameters are determined and inserted by the event manager of the recipient Web site when the event message is received.

Example 2.55 (XChange Event-Raising Rule) *The site* `http://airline.com` *has been told to notify Mrs. Smith's travel organiser of delays or cancellations of flights she travels with. The shape of such an event-raising rule is given followed by a concrete, complete XChange event-raising rule.*

```
RAISE
   <event message pattern notifying flight cancellation>
ON
   <event query detecting flight cancellations>
END
```

The following event-raising rule is registered at `http://airline.com` *and detects cancellation notification events sent by one of the airline's control points. If the flight* $AI2021$ *is cancelled, the airline notifies the organiser of Mrs. Smith about this event.*

```
RAISE
  xchange:event {
     xchange:recipient { "http://organiser.com/Smith" },
     cancellation-notification { var F }
  }
ON
  xchange:event {{
     xchange:sender { "http://airline.com/control-point20/" },
     cancellation {{
        var F -> flight {{ number { "AI2021" },
                           date { "2005-08-21" }
                  }}
     }}
  }}
END
```

2.5.3 Transaction Rules

XChange *transaction rules* are means for updating persistent data on the Web and notifying other XChange-aware Web sites of events occurring during the execution of these updates. These actions – updates and raising of events – are to be executed as a transaction. Conditions that need to hold, as a precondition for transaction execution, can be specified by means of queries to persistent data.

Transaction rules are introduced by the keyword `TRANSACTION` followed by a transaction specification, the (atomic or composite) event query is preceded by the keyword `ON`, and the Web query by the keyword `FROM`. Transaction specifications have been introduced in Section 2.4.2.3. The grammar rule defining transaction rules is given next. (*Note* that just the transaction specification is mandatory in XChange transaction rules.)

```
Trans_Rule ::= "TRANSACTION" Trans ("ON" EvQ)? ("FROM" Query)?
"END"
```

As for event-raising rules, incoming events are queried by means of event query EvQ. For each answer to EvQ the Xcerpt query $Query$ is evaluated. If $Query$ evaluates successfully, the actions specified in transaction $Trans$ are to be executed (either all of them or none). Assuming that the answers to EvQ and $Query$ contain the substitution sets Σ_{EvQ} and Σ_{Query}, respectively, for executing the specified actions (updates and events to be raised) substitution set $\Sigma = \Sigma_{EvQ} \bowtie \Sigma_{Query}$ will be used.

Note that the "event part" is not mandatory for event-raising rules and transaction rules in XChange, so as to be able to specify e.g. updates that are to be executed not (necessarily) as reaction to events.

Example 2.56 (XChange Rule for Booking a Flight) *The travel organiser of Mrs. Smith uses the following rule: if the return flight of Mrs. Smith is cancelled then look for and book another suitable flight. Again, the shape of such a transaction rule is given first.*

```
TRANSACTION
   <make flight reservation>
ON
   <event query detecting flight cancellations notifications>
FROM
   <Web query looking for another suitable flight>
END
```

The XChange rule of Example 2.55 is used to send event messages to Mrs. Smith's organiser; the next XChange transaction rule responds to it by booking another flight. If no suitable flight is found, no action is performed.

```
TRANSACTION
  in { resource { "http://airline.com/reservations/" },
       reservations {{
         insert reservation { var F, name { "Christina Smith" } }
        }}
     }
ON
```

```
  xchange:event {{
     xchange:sender { "http://airline.com" },
     cancellation-notification {{
        flight {{ number { "AI2021" },
                  date { "2005-08-21" }  }}
          }}
       }}
FROM
  in { resource { "http://airline.com" },
       flights {{
          var F -> flight {{
                      from { "Paris" }, to { "Munich" },
                      date { "2005-08-21" }
       }}
    }}
   }
END
```

Example 2.57 (XChange Rule Specifying Sequence of Updates as Action) *If no other suitable return flight is found and the airline does not provide an accommodation, then book for Mrs. Smith a cheap hotel and inform her secretary about the changes of her schedule. This is represented in XChange as a rule the travel organiser of Mrs. Smith has. The rule is shaped as follows:*

```
TRANSACTION
   <make hotel reservation>
            <and>
   <announce secretary of changes of schedule>
ON
   <event query detecting cancellations of flights for which
    the airline does not provide an accommodation>
FROM
   <Web query looking for a suitable hotel>
END
```

For a cancelled return flight from Paris to Munich, the travel organiser of Mrs. Smith uses the following XChange rule:

```
TRANSACTION
  and [
    in { resource { "http://hotels.net/reservations/" },
         reservations {{
            insert reservation {
                    var H, name { "Christina Smith" },
                    from { "2005-08-21" }, until { "2005-08-22" } }
         }}   },
     in { resource { "diary://diary/my-secretary" },
          diary {{
```

```
            news {{
              insert my-hotel {
                       remark { "I'm staying in Paris over night!" },
                       phone { var Tel },  reason { "Flight cancellation." } }
      }}  }}   }
   ]
ON
  andthen [
    xchange:event {{
       xchange:sender { "http://airline.com" },
       cancellation-notification {{
          flight {{ number { "AI2021" }, date { "2005-08-21" }  }}
       }}
    }},
    without { xchange:event {{
                 xchange:sender { "http://airline.com" },
                 accommodation-granted {{
                     hotel {{ }} }}  }}
            } during [2005-08-21T15:00:00..2005-08-21T19:00:00]
  ] within 2 hour
FROM
   result [[
       var H -> position 1 hotel {{ phone { var Tel } }} }}
     ]]

END
```

Note that the Web query (introduced by FROM*) does not query a particular Web resource; it queries a view over the data of two Web resources having different structures. The Xcerpt rule constructing the view over hotel data is given in the next section (Example 2.58). Variable H is to be bound to the hotel offering the best price.*

2.5.4 Deductive Rules

Deductive rules are means for constructing views over heterogeneous data sources. As exemplified in the previous section, data views are easily and elegantly queried in the *Web Query* part of reactive rules. Deductive rules are expressed by using the Web and Semantic Web query language Xcerpt, which is integrated into XChange.

```
Xcerpt_Rule ::= "CONSTRUCT" Construct_Term "FROM" Query "END"
```

Deductive rules of an XChange program can be chained, that is query parts of reactive or deductive rules can query the result of other deductive rules. This is realised by matching (simulation unifying) the query part with the construct part – the head – of other deductive rules. *Note* that the head of reactive rules can not be queried.

Example 2.58 (Deductive Rule for Gathering Information about Hotels) *The following Xcerpt rule queries data found at Web resources* `http://hotels.net` *and* `http://hotels-`

`paris.fr` and constructs a view over the hotel data by gathering information about all hotels in Paris. The constructed data term contains a list of hotels ordered by their price per room.

```
CONSTRUCT
 result [
    all hotel { name { var Name },
                price { var Price },
                phone { var Phone } } order by ascending [ var Price ]
   ]
FROM
 or {
    in { resource { "http://hotels.net" },
         accommodation {{
            hotels {{
               city { "Paris" },
               desc hotel {{
                       name { var Name }
                       price-per-room { var Price },
                       phone { var Phone } }} }}
         }}
     },
     in { resource {"http://hotels-paris.fr"},
          logement {{
             hotel {{
                nom { var Name },
                telephone { var Phone },
                prix { var Price }
             }}
           }}
     }
 }
END
```

Note that the two data terms queried for hotels in Paris have different structures; the above given rule not only gathers the desired information, but also gives data a uniform structure.

Complex applications specifying reactivity on the Web require a number of features that can not always be specified by simple programs. In XChange, rules are also *means for structuring* complex XChange programs.

2.5.5 Range Restriction

This section discusses *range restriction* of XChange rules, i.e. a syntactic restriction on admissible XChange rules. The satisfaction of the range restriction property by the rules of an XChange program is assumed in defining the formal semantics of XChange. Moreover, range restriction of XChange rules is a syntactical property that can be statically verified when parsing XChange programs so as to avoid (some) programming mistakes.

Intuitively, an XChange rule is range-restricted if every variable occurring in the construct term(s) of the rule's head ("action part" or construct part) has at least one defining occurrence (i.e. an occurrence that "provides" bindings for the variable) in other parts (event query part, Web query part, or actions that are to be performed before the action containing the respective construct term) of the rule.

For defining the range restriction of XChange rules, each variable occurrence in XChange rules is associated with a *polarity* and an *optionality*; these determine whether a variable occurring in a part of the rule can be used in other parts of the respective rule. A *negative* polarity (denoted by "-") of a variable occurrence expresses a defining occurrence of the variable. A *positive* polarity (denoted by "+2) expresses a non-defining variable occurrence. Optionality is given by an attribute *optional* (denoted by "?") and *not optional* (denoted by "!") for variables contained in an optional subtree and do not always have bindings.

An XChange program is a set of rules, denoted $P = \{Rr_1, \ldots, Rr_m, Tr_1, \ldots, Tr_n, Dr_1, \ldots, Dr_p\}$, where

- Rr_i, $0 \leq i \leq m$, are event-raising rules of the form $t^e \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (t^e is an event term, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Tr_j, $0 \leq j \leq n$ are transaction rules of the form $tra \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (tra is a transaction specification, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Dr_k, $0 \leq k \leq p$ are Xcerpt rules of the form $t^c \leftarrow \mathcal{Q}$ (t^c is a construct term and $\mathcal{Q}$ an Xcerpt query), and
- $1 \leq m + n$.

The range restriction of Xcerpt rules (deductive rules Dr_k, $0 \leq k \leq p$) is defined in [64], Chapter 6, pages 133-137. Thus, this section discusses only range restriction for XChange reactive rules of a program P, i.e. event-raising rules Rr_i, $0 \leq i \leq m$, and transaction rules Tr_j, $0 \leq j \leq n$. For this, the polarity of event queries, Web queries, and actions need to be defined. As Xcerpt query terms are needed for defining the polarity of event queries and updates, and construct terms are needed for defining the polarity of event terms and updates, the definition of polarities of Xcerpt subterms is given in the following; the polarity of Xcerpt query and construct terms has been defined in [64], Chapter 6.

Definition 2.4 (Polarity of Xcerpt Subterms)

1. *Let t be a query term with polarity p and optionality o.*
   - *if t is of the form* `without` t'*, then t' is of polarity $+$ (regardless of p) and optionality o*
   - *if t is of the form* `optional` t'*, then t' is of polarity p and optionality $?$.*
   - *if t is of one of the forms $l\{\{t'_1, \ldots, t'_n\}\}$, $l\{t'_1, \ldots, t'_n\}$, $l[[t'_1, \ldots, t'_n]]$ or $l[t'_1, \ldots, t'_n]$ ($n \geq 0$), then t'_1, $\ldots$, t'_n are of polarity p and optionality o.*
   - *if t is of the form* `desc` t' *then t' is of polarity p and optionality o.*
   - *if t is of the form var $X \rightarrow t'$ then t' is of polarity p and optionality o.*
2. *Let t be a construct or data term with polarity p and optionality o.*

- *if t is of the form* `optional` t', *then* t' *is of polarity* p *and optionality* ?.
- *if t is of one of the forms* $f\{t'_1, \ldots, t'_n\}$ *or* $f[t'_1, \ldots, t'_n]$ $(n \geq 0)$, *then* t'_1, ..., t'_n *are of polarity* p *and optionality* o.
- *if t is of the forms* `all` t' *or* `some` t', *then* t' *is of polarity* p *and optionality* o.
- *if t is of the form* $op(t'_1, \ldots, t'_n)$, *with op a function or aggregation identifier, then* t'_1, ..., t'_n *are of polarity* p *and optionality* o.

The root of a query term is usually of negative polarity (and thus define variable bindings) and not optional. The root of a construct or data term is of positive polarity and not optional.

2.5.5.1 Polarity of Event Queries

For assigning a polarity to each occurrence of a variable in an XChange event query, polarities are recursively assigned to each component of an event query. An XChange event query may be atomic or composite; an atomic event query is an Xcerpt query term with an optional absolute temporal restriction specification. The above given definition (taken from [64]) represents the base of defining the polarity of XChange event queries; it defines polarity of atomic event queries without temporal restrictions. The next definition is used for defining the polarity of XChange event queries.

Definition 2.5 (Polarity of XChange Event Queries) *Let eq be an event query with polarity p and optionality o. If eq is of the form:*

- *eq is an Xcerpt query term, then the Definition 2.4 is used for attributing polarity to its subterms;*
- $eq = eq'$ `in` $[b..e]$, *or* $eq = eq'$ `before` e, *then* eq' *is of polarity* p;
- $eq = eq'$ `within` w, *then* eq' *is of polarity* p;
- $eq = \texttt{and}\{eq_1, \ldots\ eq_n\}$, *then* eq_i *is of polarity* p, $1 \leq i \leq n$;
- $eq = \texttt{andthen}[eq_1, \ldots, eq_n]$ *or* $eq = \texttt{andthen}[[eq_1, \ldots, eq_n]]$, *then* eq_i *is of polarity* p, $1 \leq i \leq n$;
- $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]]$, *then* eq_i *and* q_{ij} *are of polarity* p, $1 \leq i \leq n$, $j = i + 1$, $2 \leq j \leq n$;
- $eq = \texttt{or}\{eq_1, \ldots\ eq_n\}$, *then* eq_i *is of polarity* p, $1 \leq i \leq n$;
- $eq = \texttt{var}\ X \rightarrow eq'$, *then* eq' *is of polarity* p;
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$, *then* eq_1 *is of polarity* + *(regardless of* p*) and* eq_2 *is of polarity* p;
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ [b..e]$, *then* eq_1 *is of polarity* + *(regardless of* p*);*
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ \{eq''\}$, *then* eq' *and* eq'' *are of polarity* p;
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ [b..e]$, *then* eq' *is of polarity* p;

- $eq =$ `every` n `any var` $X_1, \ldots,$ `var` X_k $\{eq'\}$, *then* eq' *is of polarity* p*;*

- $eq =$ `withrank` n `any var` $X_1, \ldots,$ `var` X_k $\{eq'\}$, *then* eq' *is of polarity* p*;*

- $eq =$ `last` $\{eq_1\}$ `during` $\{eq_2\}$, *then* eq_1 *and* eq_2 *are of polarity* p*;*

- $eq =$ `last` $\{eq_1\}$ `during` $[b..e]$, *then* eq_1 *is of polarity* p*;*

- $eq =$ (`atleast`|`atmost`)? m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $\{eq'\}$, *then* eq' *and* eq_i *are of polarity* p, $1 \leq i \leq n$*;*

- $eq =$ (`atleast`|`atmost`)? m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots, eq_n\}$ `during` $[b..e]$, *then* eq_i *is of polarity* p, $1 \leq i \leq n$*.*

Each of the component event queries of the event query eq having one of the forms given above are of optionality o.

The root of an event query is of negative polarity (it defines variable bindings) and not optional. If event exclusion is specified, the polarity changes according to Definition 2.5.

2.5.5.2 Polarity of Web Queries

Web queries in XChange event-raising rules or transaction rules are Xcerpt queries. The polarity of Xcerpt queries has been defined in [64], Chapter 6, Definition 6.2.

2.5.5.3 Polarity of Actions

XChange actions are raising of events to one or more reactive Web sites, or executing XChange transactions, i.e. ordered or unordered conjunctions or disjunctions of (elementary or complex) updates and/or raising of events.

Definition 2.6 (Polarity of XChange Event Terms) *Let et be an event term with polarity p and optionality o.*

- *if et is an Xcerpt construct term, then the Definition 2.4 is used for attributing polarity and optionality to its subterms;*

- *if et is of the form* `all` *ct, with ct Xcerpt construct term, then ct is of polarity p and optionality o.*

An XChange update term is a pattern for the data to be updated augmented with update operations. For attributing polarity to update terms, it suffices to attribute polarity to their *subjacent query terms* and the *construct terms* of the update operations (insertion and replacements). (Recall that XChange update operations cannot be nested.) The subjacent query term of an update term (Section 2.4.2) is an Xcerpt query term; thus, for attributing polarity to the subjacent query terms, the part for query terms of Definition 2.4 is used. For attributing polarity to construct terms that are part of update operations (e.g. `insert` *ct* or *qt* `replaceby` *ct*, with *ct* construct term and *qt* query term), the part for construct terms of Definition 2.4 is used.

The root of a subjacent query term is of negative polarity (it provides bindings) and not optional; the root of a construct term that is part of an update operation is of positive polarity (it consumes bindings) and not optional.

Definition 2.7 (Polarity of XChange Actions) *Let a be an XChange action specification of polarity p.*

1. *If a is an event term, then Definition 2.6 is used for attributing polarity to its subterms.*
2. *If a is an elementary update, then its subjacent query term is of polarity p and its construct terms of polarity +; the specified resources are of polarity +.*
3. *If a is of one of the forms:*
   - `and` $[a_1, a_2, \ldots, a_n]$ *or* `and` $\{a_1, a_2, \ldots, a_n\}$, *then every update a_i is of polarity p and every event term a_j of polarity +, $1 \leq i \leq n$, $1 \leq j \leq n$;*
   - `or` $[a_1, a_2, \ldots, a_n]$ *or* `or` $\{a_1, a_2, \ldots, a_n\}$, *then every update a_i is of polarity p and every event term a_j of polarity +, $1 \leq i \leq n$, $1 \leq j \leq n$.*

The root of a complex update is of negative polarity and not optional.

Example 2.59 (Polarity in an XChange Rule) *The following event-raising rule is used for settling an appointment with one of Mrs. Smith's friends. As all terms have associated attribute not optional, the optionality is not depicted in the example.*

```
RAISE
  +xchange:event {
      +xchange:recipient {"http://organiser.com/Eva"},
      +proposal {
         +text {"Can we meet?"},
         +on { +var Date }, +at {"14:00"}
      }
   }
ON
   -and {
      -without {
         +xchange:event {{
            +meeting {{
               +begin { +var Time },
               +date  { +var Date }
            }}
         }}
      } during [2005-08-21..2005-08-22],
      -xchange:event {{
         -xchange:sender {"http://organiser.com/Eva"},
         -info {{
            -text {"I'm in Munich"},
            -date { -var Date }
         }}
      }}
   } before 2005-08-22T22:00
FROM
   -in { +resource { "file:/appointments.xml" },
```

```
        -schedule {{
            -desc -appointments {{
                    -without +appointment {{
                                        +for { +var Date } }}
                    }}
        }}
    }
END
```

Note that variable `Time` *occurs only once and with positive polarity, that is* `Time` *has no occurrence that provides bindings for it. The variable* `Date` *occurs once with negative polarity, i.e. the event query against notifications from Eva provides bindings for* `Date`*; the variable has more than one occurrence with positive polarity expressing occurrences where the bindings for* `Date` *are consumed (e.g. in the event term proposing an appointment).*

2.5.5.4 Range Restriction of XChange Rules

An XChange transaction rule is range restricted if variables occurring in the construct terms of the "action part" have at least one defining occurrence in the event query, Web query, or in the updates that are to be performed before the respective variables are used. An XChange event-raising rule is range restricted if variables occurring in the event term have at least one defining occurrence in the event query or Web query.

For simplifying the definition of range restriction of XChange rules, the disjunctive normal form of an XChange rule is defined next.

Definition 2.8 (Disjunctive Rule Normal Form of XChange Rules) *An XChange rule* $A \leftarrow_r Q \leftarrow_r Eq$ *is in disjunctive normal form if* Q *and* A *are in disjunctive normal form, that is*

(i) an Xcerpt query Q *is in disjunctive normal form if it is of the form* $\bigvee_j q_j$*, where* q_j *is a conjunction of query terms and/or negated query terms;*

(ii) an action A *is in disjunctive normal form if it is an event term or a disjunction of actions, where each action is a conjunction of elementary updates and/or event terms; the conjunction of actions preserves the order of action specifications.*

Bringing a Web query in its disjunctive normal form is rather straightforward and has been discussed in [64], Chapter 6. The steps followed for transforming an XChange action specification into its disjunctive normal deserve some explanation. For bringing an action specification A in its disjunctive normal form,

1. each *ordered disjunction* of A is transformed into an unordered one; this step does not influence the bindings of the variables occurring in the actions specified in the disjunction (recall that component actions of an unordered or ordered disjunction "share" only the variable bindings obtained from evaluating the event query and the Web query). Thus, actions of the form `or`$[a_1, a_2, ..., a_n]$ are transformed into `or`$\{a_1, a_2, ..., a_n\}$.

2. each *unordered conjunction* of action specification of A is transformed into a disjunction of ordered conjunctions. For example, an unordered conjunction of the form `and`$\{a_1, a_2\}$ is transformed into

   `or`{

$\texttt{and}[a_1, a_2]$,

$\texttt{and}[a_2, a_1]$ }

Note that the transformation preserves the definition of the scope of variables in unordered conjunction of actions (cf. Section 2.4.2.2).

3. the last step in the transformation consists in placing the action specification resulted from applying 1. and 2. to A into *disjunctive normal form*; this is achieved by recursively moving conjunctions inward and disjunctions outward by using the rewriting rules:

   $\texttt{and}[a, \texttt{or}\{b, c\}] = \texttt{or}\{\texttt{and}[a, b], \texttt{and}[a, c]\}$

   $\texttt{and}[\texttt{or}\{a, b\}, c] = \texttt{or}\{\texttt{and}[a, c], \texttt{and}[b, c]\}$

   By applying the above stated rules, the order of actions in ordered conjunction specifications needs to be preserved.

For defining the range restriction of XChange rules, a predicate $\texttt{prob}(eq, V)$ is defined for expressing that the event query eq provides bindings for the variable V occurring in eq. If $\texttt{prob}(eq, V) = True$, V can be consumed in other parts of the rule.

Definition 2.9 (Predicate `prob` - Provides Bindings) *Let eq be an event query and V a variable occurring in eq. The predicate* $\texttt{prob}(eq, V)$ *is defined recursively on the structure of eq.* $\texttt{prob}(eq, V) = True$*, iff*

- *eq atomic event query and V occurs with negative polarity in eq;*
- $eq = eq'$ `in` $[b..e]$*, or* $eq = eq'$ `before` e*, and* $\texttt{prob}(eq', V) = True$*;*
- $eq = eq'$ `within` w *and* $\texttt{prob}(eq', V) = True$*;*
- $eq = \texttt{and}\{eq_1, \ldots\ eq_n\}$ *and* $\exists i$*,* $1 \leq i \leq n$ *such that* $\texttt{prob}(eq_i, V) = True$*;*
- $eq = \texttt{andthen}[eq_1, \ldots, eq_n]$ *or* $eq = \texttt{andthen}[[eq_1, \ldots, eq_n]]$*, and* $\exists i$*,* $1 \leq i \leq n$ *such that* $\texttt{prob}(eq_i, V) = True$*;*
- $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]]$ *and* $\exists i$*,* $1 \leq i \leq n$ *such that* $\texttt{prob}(eq_i, V) = True$ *or* $\texttt{prob}(q_{ij}, V) = True$*;*
- $eq = \texttt{or}\{eq_1, \ldots\ eq_n\}$ *and* $\forall i$*,* $1 \leq i \leq n$*:* $\texttt{prob}(eq_i, V) = True$*;*
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$ *and* $\texttt{prob}(eq_2, V) = True$*;*
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ \{eq''\}$ *and* $\texttt{prob}(eq', V) = True$ *or* $\texttt{prob}(eq'', V) = True$*;*
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ [b..e]$ *and* $\texttt{prob}(eq', V) = True$*;*
- $eq = \texttt{every}\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}$ *and* $\texttt{prob}(eq', V) = True$*;*
- $eq = \texttt{withrank}\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}$ *and* $\texttt{prob}(eq', V) = True$*;*
- $eq = \texttt{last}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$ *and* $\texttt{prob}(eq_1, V) = True$ *or* $\texttt{prob}(eq_2, V) = True$*;*

- $eq = \texttt{last}\ \{eq_1\}\ \texttt{during}\ [b..e]$ *and* $\texttt{prob}(eq_1, V) = True$;
- eq = $(\texttt{atleast}|\texttt{atmost})?\ m\ \texttt{of any var}\ X_1, \ldots \texttt{var}\ X_k\ \{eq_1, \ldots eq_n\}\ \texttt{during}\ \{eq'\}$ *and* $\texttt{prob}(eq', V) = True$ *or* $\forall i,\ 1 \leq i \leq n$: $\texttt{prob}(eq_i, V) = True$.

For all other cases, $\texttt{prob}(eq, V) = False$. *Actually, predicate* `prob` *states whether a variable can be considered as having negative polarity in a (composite) event query.*

Every XChange rule can be transformed into disjunctive normal form. Based on this result and using the predicate defined above, the range restriction of XChange rules can be formalised by the following definitions:

Definition 2.10 (Range Restriction of XChange Event-Raising Rules) *Let R be an XChange event-raising rule and $R' = t^e \leftarrow_r \bigvee_j q_j \leftarrow_r Eq$ its disjunctive normal form. R is range restricted iff*

- *forall variables V occurring in t^e with positive polarity, V occurs in Eq such that* $\texttt{prob}(Eq, V) = True$ *or in each of the q_j with negative polarity;*
- *forall variables V occurring in Eq or in at least one q_j with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq and q_j, V is also attributed as optional in all occurrences in Eq, q_j, and t^e.*

Definition 2.11 (Range Restriction of XChange Transaction Rules) *Let R be an XChange transaction rule and $R' = \bigvee_k a_k \leftarrow_r \bigvee_j q_j \leftarrow_r Eq$ its disjunctive normal form. R is range restricted iff*

- *forall variables V occurring in one of the construct terms of a_k (that is, in one of the update or event terms of a_{kl}, where $a_k = \bigwedge_l a_{kl}$) with positive polarity, V occurs with negative polarity in Eq and/or in each of the q_j, and/or occurs with negative polarity in at least one of the a_{kl}, $1 \leq l \leq p$, where V in a_{kp};*
- *forall variables V occurring in Eq or in at least one q_j with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq and q_j, V is also attributed as optional in all occurrences in Eq, q_j, and a_k;*
- *forall variables V occurring in the subjacent query term of an update term a_{kl} with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq, q_j, and a_{kw} with $1 \leq w \leq l-1$, V is also attributed as optional in all occurrences in the construct terms of a_{kl}.*

An XChange program P is range restricted if all rules $R \in P$ are range restricted.

2.6 Implementation of Use Cases with XChange

We end this Chapter by illustrating XChange on a selected subset of the use cases defined in our previous deliverable [3]. This subset is also used in Appendix A for illustrating the ruleCore framework. We have chosen use-cases from a single scenario, the "REWERSE Information System Portal" [3]. We briefly recall that in this scenario we consider nodes for:

- participants of the REWERSE project, containing information on their names, country, url, role in the project, etc;
- working groups, containing information name, id, coordinator and deputy coordinator, deliverables, participants involved, etc;
- one central node of the project containing general information, and references to the working groups and participants.

For this scenario, several use cases of reaction to changes in nodes, and evolution and propagation of changes are defined. Here we use a selection of them to illustrate:

Propagation of updates: for this we selected Use-Case 4.2.1 (Changing Phone Number), extending it with a variation Use-Case 4.2.1a (Changing Roles) so that the use of transactions may also be illustrated at this basic level;

Dissemination of information: different strategies for propagation and distribution of information, through Use-Case 4.2.7 (Progress Reports) that illustrates both *pull* and *push* strategies;

Composite events: the notion of event algebras is illustrated by Use-Case 4.2.13 (Progress Report Late), and is complemented with Use-Case 4.2.14 (Polls: Basic Rules).

Further details on the specification of the use cases are to be found at [3].

2.6.1 Changing and Propagating Contact Information

Use Case 4.2.1 (Changing Phone Number) Phone numbers (or any other contact details) are updated at the participants nodes (XML format). The updates have to be propagated to the WG nodes and to the central node.

The following rule runs locally at the participant node `http://.../goettingen.xml`. It detects (implicit) events signaling a change in contact information and updates the working group node resource "http://.../workinggroup-i5.xml" accordingly.

```
TRANSACTION
    in { resource {"http://.../workinggroup-i5.xml"},
        desc person {{
            attributes {{ id { var ID } }},
            contact {{ }} replaceby var C
        }}
    }
ON
    xchange::event {{
        updated-contact {{
            person {{
                attributes {{ id { var ID } }}
            }}
        }}
    }}
FROM
```

```
    in { resource {"http://.../goettingen.xml"},
        desc person {{
            attributes {{ id { var ID } }},
            var C -> contact {{ }}
        }}
    }
END
```

A disadvantage of this rule is that the working group node is "hard-coded." We can make the rules more flexible by assuming a file `"http://rewerse.net/workinggroups.xml"` that lists all working groups and the URIs of their homepages:

```
TRANSACTION
    and {
        all in { resource { var H },
            desc person {{
                attributes {{ id { var ID } }},
                contact {{ }} replaceby var C
            }}
        }
    }
ON
    xchange::event {{
        updated-contact {{
            person {{
                attributes {{ id { var ID } }}
            }}
        }}
    }}
FROM
    and {
        in { resource {"http://.../goettingen.xml"},
            desc person {{
                attributes {{ id { var ID } }},
                var C -> contact {{ }}
            }}
        },
        in { resource {"http://rewerse.net/workinggroups.xml"},
            desc workinggroup {
                homepage { var H }
            }
        }
    }
END
```

Instead of detecting an implicit event that signals a change in the contact information, we can also have an explicit event (sent e.g., via an HTML form with HTTP POST) *request* a change of contact information. The following rule receives such a request and updates both the working group and R node:

```
TRANSACTION
    and {
        in { resource { var R }
            desc person {{
                attributes {{ id { var ID } }},
                contact {{ }} replaceby C
        },
        in { resource { "http://.../workinggroup-i5.xml" },
            desc person {{
                attributes {{ id { var ID } }},
                contact {{ }} replaceby C
            }}
        }
    }
ON
    xchange:event {{
        change-of-contact-information {{
            resource {{ var R }},
            var N -> name {{ }},
            var C -> contact {{ }}
        }}
    }}
FROM
    in { resource { var R},
        desc person {{
            attributes {{ id { var ID } }},
            var N
        }}
    }
END
```

2.6.2 Change of Role for a Coordinator or Deputy Coordinator

Use-Case 4.2.1a (Changing Roles) The `role` (`coordinator`, `deputy coordinator` or `assistant`) assigned to the `participation` of a `person` in a working group may be updated at the participants' node.

For this use case, we consider a variation where a person requests to change his or her role attribute to `deputycoordinator`. This information is propagated from the participant node to the working group nodes and the project node. The following constraint needs to be observed: a person can only be coordinator or deputy coordinator of one working group at a time and each working group has at most one coordinator and at most one deputy coordinator.

```
TRANSACTION
    and {
        in { resource { var R },
            desc person {{
                attributes {{
```

```
                    id { var ID },
                    insert role { var NR }
                }}
            }}
        },
        in { resource { var W },
             resource { http://rewerse.net/project },
            workinggroup {{
                id { var P },
                deputycoordinator {{ }}
                    replaceby deputycoordinator {{ var G, var F}}
            }}
        }
    }
ON
    xchange:event {{
        change-of-role {{
            resource {{ var R }},
            person-id { var ID },
            participation { var P },
            new-role { "deputycoordinator" }
        }}
    }}
FROM
    and {
        in { resource { var R },
            desc person {{
                attributes {{ id { var ID } }},
                name {
                    givenname  { var G },
                    familyname { var F }
                }
            }}
        },
        not in { resource { http://rewerse.net/project.xml },
            desc deputycoordinator { var G, var F }
        },
        not in { resource { http://rewerse.net/project.xml },
            desc coordinator { var G, var F }
        },
        in { resource { http://rewerse.net/workinggroups.xml },
            desc workinggroup {{
                id { var P },
                homepage { var W }
            }}
        }
    }
```

```
END
```

2.6.3 Progress reports

Use Case 4.2.7 (Progress Reports) The deadline for the progress report is inserted into the central node, and then communicated to the WGs nodes. From there, the persons are called to send input (by mail, probably 10 days before the deadline), and the coordinator is called to produce the report. The coordinator then puts the report in the WG node. An active rule then publishes the report on the WGs Web page, and removes the deadline entry from the WG node and from the coordinators person entry in the participants node. Depending on push or pull strategy, (i) the WG node sends the report to the central node, or (ii) the central node reacts on the remote event on the WG node.

For implementing this use case, rules must be added at the levels of the working group nodes, participant nodes and central node.

At the working group node:

```
TRANSACTION
    and {
        in { resource { "http://.../workinggroup-i5.xml" },
            workinggroup {{
                progress-report {
                    delete deadline {{ }},
                    insert var R
                }
            }}
        },
        all xchange:event {
            xchange:recipient { var T },
            var P
        },
        xchange:event {
            xchange:recipient { "http://rewerse.net/project" },
            var P
        }
    }
ON
   var P -> new-progress-report {{
        coordinator { var C },
        workinggroup { var W },
        var R -> report {{ }}
    }}
FROM
    in { resource { http://rewerse.net/workinggroups.xml },
        desc workinggroup {{
            id { var W },
            participants {{
                participant {{
                    homepage { var T }
```

```
                }}
            }}
        }}
    }
END
```

At the participant nodes:

```
TRANSACTION
    in { resource { "http://.../goettingen.xml" },
        desc person {{
            attributes { id {"C"} },
            delete deadline {{
                type { "progress-report" }
            }}
        }}
ON
   xchange:event {{
       new-progress-report {{
          coordinator { var C }
       }}
   }}
END
```

And at the central node:

```
RAISE
   xchange:event {{
       xchange:recipient { "mailto:schwertel@rewerse.net" },
       content [
          "Hi Uta,",
          "  All progress report have been received!",
          "Regards,",
          "  XChange"
       ]
   }}
ON
   and {
       xchange:event {{
           new-progress-report {{
               number { "2005-1" },
               workinggroup { "I1" }
           }}
       }},
       xchange:event {{
           new-progress-report {{
               number { "2005-1" },
               workinggroup { "I2" }
           }}
```

```
        }},

        ...

        xchange:event {{
            new-progress-report {{
                number { "2005-1" },
                workinggroup { "I5" }
            }}
        }}
    }
END
```

2.6.4 Late progress reports

Use Case 4.2.13 (Progress Report Late) For each progress report, a deadline is specified when it must arrive, e.g., <todo type="progressreport" number="1-2005" deadline="28.2.2005"/> Then, there is a rule that states that if for any WG, the progress report has not been checked in until noon at the day of the deadline, a message is sent to the WGs coordinator.

```
RAISE
    xchange:event {
        xchange:recipient { "mailto:coordinator-i5@rewerse.net" }
    }
ON
    xchange:event {{
        xchange:type { "timer" },
        time { "2005-02-28T12:00" }
    }}
FROM
    in { resource { "http://rewerse.net/project.xml" },
        desc progress-report {{
                    deadline {{ "2005-02-28" }},
                    workinggroup { "I5" }
                    without report {{ }}
        }}
    }
END
```

2.6.5 Polls

Use Case 4.2.14 (Polls: Basic Rules) We consider the following task: the project office makes a poll where an answer is needed from each WG coordinator. Then, there are different ways how to deal with it. Simple rules apply for making the poll known to the relevant persons (push to the participants' nodes or pull by them, or a rule at the central node that simply sends a mail to the people). Incoming responses are stored in the database as contents of the poll element; they are evaluated after the deadline.

```
TRANSACTION
    and {
        in { resource { "http://rewerse.net/polls.xml" },
            polls {{
                insert var P
            }}
         },
         all xchange:event {
             xchange:recipient { var T },
             poll {
                 id { var PID },
                 var Q
             }
         }
    }
ON
    xchange:event {{
        new-poll {
           attributes { id{ var PID } },
           var P -> poll {{
               persons {{ var T }},
               var Q -> question {{ }}
           }}
        }
    }}
END

TRANSACTION
    in { resource { "http://rewerse.net/polls.xml" },
         polls {{
             poll {{
                 attributes { id { var PID } },
                 insert answer {
                     var F,
                     var X
                 }
             }}
         }}
    }
ON
    xchange:event {{
        answer { poll-id { var PID },
                 var F -> person {{ }},
                 var X -> text {{ }},
               }
    }}
END
```

Chapter 3

A General Framework for Evolution and Reactivity in the Semantic Web

As already mentioned in the introduction, a main goal of the *Semantic Web* is to deal with the heterogeneity of data formats and languages in the Web and provide unified view(s) on the Web, as an extension to today's *portals*. In contrast to the current Web, the *Semantic Web* should be able not only to support querying, but also to propagate knowledge and changes in a semantic way. This *evolution* and *behavior* depends on the cooperation of nodes. This support for propagation of knowledge and changes, and the ability react on happenings in the Web, is already provided by the XChange language presented in the previous chapter.

However, in the same way as the goal of the *Semantic Web* is to bridge the heterogeneity of data formats, schemas, languages, and ontologies used in the Web to provide semantics-enabled unified view(s) on the Web, the heterogeneity of concepts for expressing behavior requires for an appropriate handling on the semantic level. When considering dynamic issues, the concepts for describing and implementing behavior will surely be diverse, due to different needs, and it is unlikely that there will be a unique language for this throughout the Web. Since the Web nodes are prospectively based on different concepts such as data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that the *concepts* and the actual *languages* are independent. As such, besides having a concrete language for dealing with evolution and reactivity, the Semantic Web calls for the existence of a framework able to deal with this heterogeneity of languages.

In this respect, *reactivity* and its formalization as *Event-Condition-Action (ECA) rules*, as in XChange, provide a suitable common model because they provide a modularization into clean concepts with a well-defined information flow. It is our stance that ECA rules provide a generic uniform framework for specifying and implementing communication, local evolution, policies and strategies, and altogether global evolution in the Semantic Web.

In this chapter, we propose an ontology-based approach for describing (reactive) behavior in the Web and evolution of the Web that follows the ECA paradigm. We propose a modular framework for *composing* languages for events, conditions, and actions by separating the ECA semantics from the underlying semantics of events, conditions and actions. This modularity

allows for high flexibility wrt. the heterogeneity of the potential sublanguages, while exploiting and supporting their meta-level *homogeneity* on the way to the Semantic Web.

Remember that in the above presentation of XChange each of the three components of ECA rules was presented separately, and in fact each part uses a (sub)language that in principle could be combined with other sublanguages. Such a modular framework allows for combining exactly the three (sub)languages of XChange (with an appropriate treatment of communication between these (sub)languages by the generic communication between the various parts of an ECA rule, as will be developed), but also for combining them with other languages for events, for querying, and for executing actions including updating Web data. This way, it should be possible to write a rule that reacts on events that are monitored by a system like the ruleCore system described in Appendix A, then grabs some extra data using e.g. XQuery, perform some tests on the data, and possibly executes some actions of updating some XML data using the sophisticated mechanisms of XChange for this purpose. So far, these rules handle distributed XML data on the Web.

Another important aspect when considering the *Semantic Web* is that of abstraction levels of behaviour. As it will be detailed below, it is our opinion that evolution and reactivity will appear at several levels in the Semantic Web: there will be local basic events and actions, similar to local database triggers; events on local XML data; global rules in XML data that are able to react on views that include remote data; and application-level events and rules referring to the terms of the ontology of an application. The XChange language mainly deals with events that are communicated (in a push strategy) from outside but are then dealt with locally; also actions are either updates of XML data, or the raising of events. In the general framework below we propose to deal also with different levels of behaviour.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general ECA rules (and their components) must be communicated between different nodes, and may themselves be subject to being updated; also reasoning about evolution might be desired. For that, the ECA rules themselves must be represented as objects of the (Semantic) Web, adhering to an own ontology of rules, and marked up in an (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [63], but it does not tackle the complexity and language heterogeneity of events, actions, and the generality of rules, as described here. In this report we sketch a markup for active rules, that will be the basis for a (near) future discussion with the WPI1 of REWERSE (Rule Markup) in order to establish the final markup proposal.

Structure of the chapter. We start the chapter by summarizing some requirements posed by evolution and reactivity on the *Semantic Web*. Here, after some brief points on the important differences between dealing with the level of the Web versus at the level of the Semantic Web, we elaborate on the various abstraction levels of behaviour that need to be considered in the semantic level. This leads a discussion of how to extend the domain ontologies with actions and events, and the various types of events that need to be considered in a general framework. Then, we summarize what kinds of rules, according to their tasks in the framework, have to be covered. Section 3.2 ECA rules on the lowest level, namely triggers on the data model level.

We then discuss the main issue of dealing with the above mentioned heterogeneity of languages. For this we start, in Section 3.3, by proposing a general structure, rule level ontology and corresponding markup, for (ECA) reactive rules. In it, basically each rule consists of:

- An *event part*, in which there must be a description of something that, if it happens, fires the rule;

- A *condition part*, in which, depending on the detected event, may collect some further (static) data and test conditions on both the event and the collected information to certify that an action has actually to be executed. Accordingly, this condition part can be further decomposed into one or more *query parts* for collecting data, and one *test part* for the checking the condition;
- An *action part*, describing what to do when the event is detected and the condition test succeeds.

The components of a rule use different sublanguages for expressing events, queries, tests or actions. Not only the sublanguages of each family share some properties, but there are common properties of all kind of such sublanguages. We analyse the structure of these languages from the semantical and structural aspects and summarize their common aspects: they are algebraic languages, consisting of nested expressions.

Coming back to the rule level, one essential point here is how these parts communicate with each other. As for XChange, here we also propose the usage of logical variables for this purpose. Various aspects concerning the definition and usage of these variables are discussed in Section 3.4. We then proceed in Section 3.5 with defining the general overall semantics of execution for these rules, including the semantics of execution of expressions in each of the parts. In Section 3.6 we illustrate each of the Event/Query/Test/Action components with some (sample) concrete languages. Here we focus on the event component by describing how the event algebra similar to that of SNOOP [25] is mapped into our framework. A Web-Service-oriented architecture of the general framework in the Semantic Web based on the "actual" *resources* (e.g., language processors that are associated with the (sub)language resources) of the RDF ontology is then described in Section 3.7. Section 3.8 shortly sketches some obvious optimizations and further research issues. Considerations on the planned implementation are fixed in Section 3.9, including the architecture of the implementation, and an evaluation of possible already existing component languages. Section 3.10 gives a brief comparison with the existing languages for active rules for XML and RDF, including the XChange language described in the previous chapter, and show how they are covered in our framework, and where they can be useful in the architecture.

A preliminary version of the general framework described in this chapter has been published in [1], the ontology of rules, rule components and languages, and the service-oriented architecture proposal have been published in [50], and the languages and their markup, communication and rule execution model can be found in [49].

Issues that we are not dealing with. There are several issues that are explicitly not dealt with in our approach – because they are encapsulated inside (and "bought with") the concepts to be integrated: The detection of composite events is done and provided by the individual event languages and their engines – we are only providing an environment for embedding them. In the same way, query evaluation itself is left to the original languages and processors to be embedded into the global approach. Actual execution of actions (and transactions) is also left with the individual solutions.

3.1 Requirements Analysis

This section analyses the requirements for ECA-based evolution and reactivity in the Semantic Web and sets some working hypotheses. We investigate the abstraction levels used in the

Semantic Web and its infrastructure, the structure of domain ontologies when dealing with dynamic aspects, then have a more detailed look on the kinds of events that have to be modeled, and classify several kinds of rules that are then actually needed. Finally we "initialize" the way towards our approach by discussing *triggers* as very simple ECA rules and illustrate why this approach provides a good base, but that a compherensive framework must extend this idea in many aspects.

3.1.1 Web vs. Semantic Web

Whereas in the conventional XML/HTML Web, ECA models and languages that operate on the data level and on explicit events are sufficient, the situation for a Semantic Web framework is much more complex:

- Model (RDF): With RDF, the same resource can be described at different physical locations, using its URI. Thus, changes in the description of "something" are not necessarily located at a given node.
- Model (OWL): While some research has already been done in the area of queries and static reasoning on the OWL level, the extension to events and actions is still completely open. Domain ontologies must define their (derived) atomic events in terms of changes to the underlying data, and, in case in addition to just execution of rules, *reasoning* is intended, also actions must be described in terms of their effects on the data.

  Developing an approach and case-studies for this has been identified as an important task for understanding what functionality and expressiveness should be provided by languages for describing behavior.
- Model and Languages: Rules in the Semantic Web exist on different abstraction levels (see Section 3.1.2) and should "cover" existing approaches. For this, composite events, conditions involving several nodes, and complex actions must be supported. Here, the existing and expected future heterogeneity has to be taken into account.
- Model, Languages and Architecture: Rules are themselves part of the Semantic Web. For this, they have to be seen as resources. On a smaller granularity, rule components and smaller identifiable parts like individual event descriptions are also resources. Rules and rule components have to be described not only syntactically in terms of a *programming language*, but on the *(rule) ontology level* which is then *translated* to actual, executable specifications in one or more programming languages.
- Languages and Architecture: Languages are resources (that have to be described by a suitable ontology). From this point of view, semantics and processors that implement this semantics are also resources and have to be described and correlated on the ontology level.

3.1.2 Abstraction Levels

Data Model Abstraction Levels. As described above, the *Semantic Web* can be seen as a network of autonomous (and autonomously evolving) nodes. Each node holds a *local* state consisting of extensional data (facts), metadata (schema, ontology information), optionally a knowledge base (intensional data), and, again optional, a behavior base. In our case, the latter is given by the ECA rules under discussion that specify which actions are to be taken upon which events under which conditions.

According to the Semantic Web Tower, there are already from the static point of view several abstraction levels.

In classical database systems, the *physical level/model/schema*, the *logical level/model/schema* (i.e., an *abstract data type* that can have different implementations/physical models, and as a database model comes with a generic *database query language*), and sometimes the *export schema* are distinguished.

In the early *network data model*, there was only the physical model where the query language constructs were also directly based on. For *relational databases*, the physical model includes the tables (i.e., ordered sets) and storage data structures (including indexes etc.), the logical model is the relational/SQL schema, and the export schema is in most cases also a relational schema, including views.

With *object-oriented* databases, there came different physical models, including relational and "native" storage. The logical model is the object-oriented model with an ODL/OQL schema; the export model is also the object-oriented one. *Object-relational* architectures are those where the physical model is the relational one, the logical model is split into two levels, the low-level one is the relational model, and the high-level is the object-oriented model, which also serves as export model.

With *XML*, the relationships became even more complex. There are several physical models that can serve for XML data; one of them is again the relational one. XML data can also be stored in object-oriented structures, as done in the early products Tamino (based on Adabas) and Excelon (based on Object Store). Several products claimed to use a "native" XML data model. The "lowest" well-specified XML-related notion is then the *Document Object Model (DOM)* [29] as an *abstract datatype*. Since this model is only on the level of an abstract datatype and does not support any query language, it is not a logical data model. The *logical data model* then is XML, with query languages like XQuery. On the other hand, XML serves as an *export data model* when XML views are defined over relational data [31].

In the research community, models like OEM [56] or F-Logic [42] (for which several internal physical models can be used, e.g., a frame-based one, or a relational one together with Datalog) came up that are used as logical models, or as export models for integrating data from other models.

Adding more abstraction with *RDF*, RDF is seen as the *logical data model*. Then, between it and –numerous– physical models, there can be a layer that uses the XML data model or the relational data model. Using "native" RDF databases, the physical data model can be any data structure that stores triples, or a frame-based structure like F-Logic. When exporting or integrating relational or XML data in RDF, RDF serves as export model.

Considering *OWL*, it qualifies as an *export model* since –by its reasoning– it defines views over RDF data as *logical model* that are queried by the user. From the point of view of the OWL/RDF user, XML then is not a logical model, but "below" this.

In general, the user works on the *export level* (which uses in general a data model which is the same or closely related to the *logical level*). Queries against the export level are mapped down to the logical level.

Abstraction Levels in the Conventional Web and in the Semantic Web. For the conventional (XML) Web and for the Semantic Web, there are different "towers" of data models: In the conventional Web, there are two levels; the upper of which is XML:

- Data level. Files, SQL databases, XML databases etc. Here *local* behavior of the *databases*

(e.g., integrity maintenance) is located.

- Logical Level: XML. Here *local behavior* of the *nodes* (e.g., local application behavior) is located. Remote actions between tightly coupled nodes (i.e., that use common XML Schemas etc.) are also possible on this level. Interfaces for Web-Services like SOAP are also on this (syntactical) level.

In the Semantic Web, the structure of the levels has to be seen from a local and from a global aspect:

- Data level. Files, SQL databases, XML databases etc. Here again *local* behavior of the *databases* (e.g., integrity maintenance) is located.
- Local Logical Level: this level is provided by an XML or relational model, sometimes omitted (for RDF databases). Here *local behavior* of the *nodes* (e.g., local application behavior) is located. Remote actions between tightly coupled nodes (i.e., that use common XML Schemas etc.) are also possible on this level.
- Global Logical/Integrated Level: RDF. Here, *integrated* behavior (i.e., simple push/pull communication) will be located; messages between loosely coupled nodes that communicate in an application domain will be exchanged on this level.
- Semantic Level: OWL. Here, *intelligent integrated behavior* will be located, i.e., business rules, policies and strategies that often use derived data (and derived events).

Abstraction Levels of Behavior. In the same way as there are different levels from the static point of view, behavior can be distinguished wrt. these levels (programming language/-data structure level, logical level, integrated level, and semantic level), and with different scope (local or global).

On all these levels, there is behavior. The user and the applications rely on the behavior on the *export model level* (OWL), whereas the actual, persistent changes to the database take place on the *physical level* (SQL, XML). On this lowest level, several proposals for "triggers" for XML data analogous to the SQL triggers exist, where with the distributed environment of XML data on the Web now, two types can be distinguished:

- Local triggers where event, condition, and action components use only the local database (like for SQL triggers),
- "Web-Level triggers" whose event component is based on data-level events in the local database, the condition component uses the local database and possibly also remote ones, and the action component can include arbitrary actions on the Web level (sending messages, SOAP),

Such trigger concepts for XML and RDF as simple ECA rules will be investigated in more detail in Section 3.2. A comparison of the framework presented in this work with existing languages will be given in Section 3.10.

For realizing behavior in the Semantic Web, also vertical transmission between the levels is required, including the XML and RDF models. A classification of types of rules wrt. their functionality and roles in the whole framework will be given in Section 3.1.5.

3.1.3 Domain Ontologies including Dynamic Aspects

The coverage of *domain ontologies* differs already in the classical data models (and conceptual models): in the relational model and in the Entity-Relationship model, a domain ontology

consists only of the static notions, expressed by relations and attributes, or entity types with attributes and relationships (similar in first-order logic). In the object-oriented model and in UML, the static issues are described by classes, properties and relationships, and the dynamic issues are described by actions; in UML also their effects can partly be described.

A *complete* ontology of an application domain requires to describe not only the static part, but also the dynamic part, including actions and events (cf. Figure 3.1):

- describing actions in terms of agents, preconditions, and effects/postconditions,
- describing events, i.e., correlating actions and the resulting events, and specifying composite events, and
- describing composite actions (processes),
- ◦ in fact, business rules themselves can also be seen as parts of the ontology of an application.

For designing a Semantic Web application or service, in general ontologies of several domains interfere:

- Application domain ontologies define the static and dynamic notions of the application domain (banking, travelling, etc.), i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of train schedule changes, actions of reserving tickets).
- Application-independent domains that *talk about* an application; mostly related to classes of services (messaging, transactions, calendars, generic data manipulation). They can be generically used in combination with arbitrary application domains. They also provide static and dynamic notions.

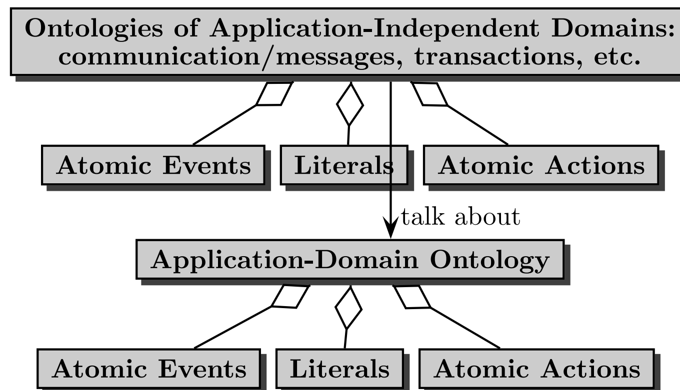


Figure 3.1: Kinds and Components of Ontologies

3.1.4 Events

An important aspect is the analysis of types of events that have to be considered. The ontology of events has to consider the abstraction levels and the application-dependent and application-independent domain ontologies. There are different kinds of (atomic) events (see explanations below and Figure 3.2):

- events of a given application domain (e.g., in banking, travel organizing, administration); such atomic events are described in terms of the ontologies of the application domain,
- generic parametric events that are not from any specific application domain but that instantiate generic event patterns, e.g., communication ("receive a message about ...") and transactional events that *talk about* application domains.

  Data level events are also a special kind of such generic (= generic to the data model) events.

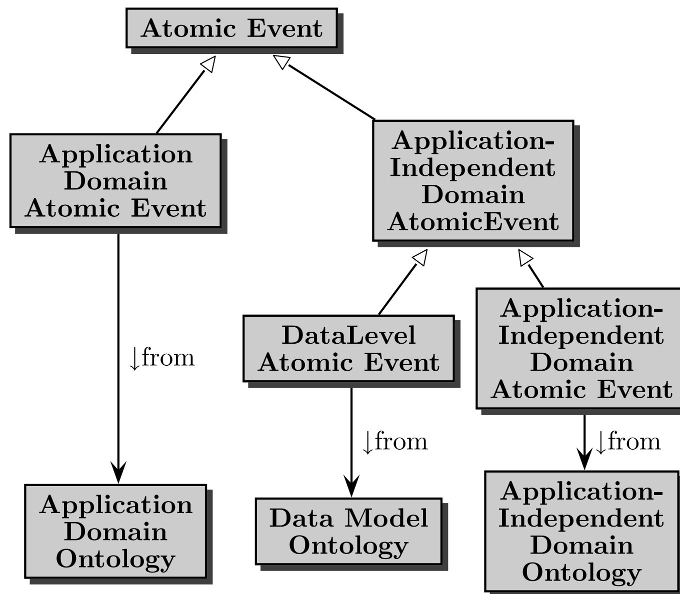


Figure 3.2: Ontology of Atomic Events

We start with the conceptually simpler events in application-independent domains.

Events in Application-Independent Domains. The application-independent domains provide *patterns* of atomic events that are ontologically independent from the actual application, but *talk about* an application; mostly related to classes of services, e.g., messaging or operations in a data model (cf. Figure 3.3). In general, such events are associated with a certain node.

Data Model Events. In the same way as for SQL data, there are atomic generic data model-level (i.e., XML or RDF) events (as will be discussed in more detail Section 3.2 for triggers). The actions that raise such events are operations of the underlying data model. Thus, they are *generic* in the sense that they apply to the schema level of a given data model and only their parameters, i.e., names of classes and relationships and actual data, are taken from the application.

Other Generic Events. In the same way as the above data model events, there are *generic* events that are not raised by data model-level updates but belong to high-level application-

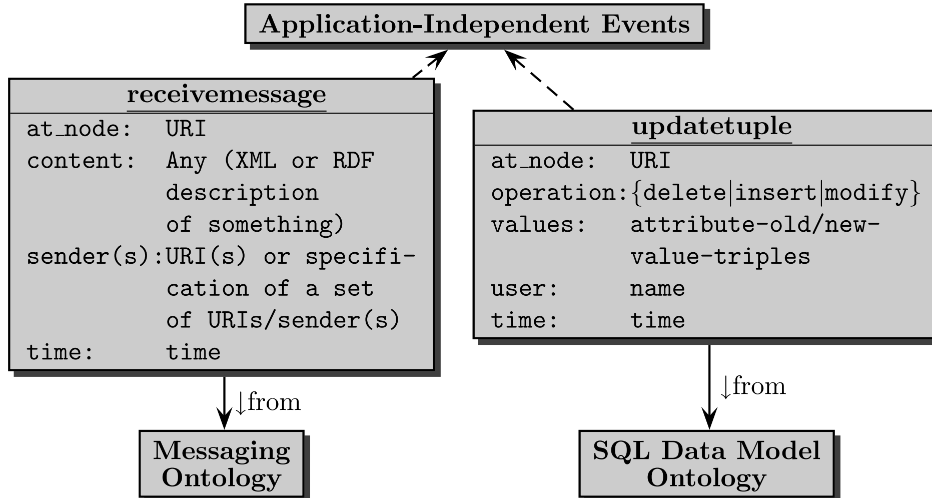


Figure 3.3: Example Events of Application-Independent Domains

independent ontologies that often *deal* with application-specific information:

- Communication: receiving or sending messages. Such events are parametric with e.g., sender and receiver (as URIs) and application-specific data (content of the message, possibly the roles of sender/receiver in this communication).
- Transactions: start, commit, and rollback of transactions. E.g., the "decision" or the message from another node that a transaction cannot be executed successfully raises such an event.

In the following, we also call such events *generic parametric events*.

Application Domain Events. Atomic application domain events are the visible happenings in the application domain. High-level rules, e.g., *business rules* use application specific events (e.g., professor_hired($object, $subject, $university)). Such events must be described by the ontology of an application.

Actions vs. Events. In contrast to simple data-level events on XML or RDF data, on the application-level there is an important difference between *actions* and *events*: an event is a visible, possibly indirect or derived, consequence of an action. For instance, the action is to "debit 200E from Alice's bank account", and visible events are "a debit of 200E from Alice's bank account", "a change of Alice's bank account" (that is also immediately detectable from the update operation), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update). Another example is the action "book person P on flight LH123 on 10.8.2005" which results in the internal action "book person P for seat 42C of flight LH123 on 10.8.2005" and the events "a person has been booked for a seat behind the kitchen", "flight LH123 on 10.8.2005 is fully booked", "there are no more flights on 10.8.2005 from X to Y", or "person P has now more than 10.000 bonus miles". All these events can be used for formulating (business) rules.

Reaction to Events. For actually reacting on events, a node must be *aware* of them. Here, some issues have to be considered in the Semantic Web since events can be derived ones, and that they are not necessarily located at a certain node.

- Explicit events are events that have a *direct* relationship with an action (and, from the implementation point of view, with a code fragment where they can be "caught"). The database-level events are obviously explicit ones: an update of the database is an action *and* is seen as an event. The receipt of a message is also an explicit event (although originally resulting from a different (sending) action, it can be caught on the socket level of a node).
- Implicit and derived events: an event is derived if it is defined based on other events, e.g. "flight LH123 on 10.8.2005 is fully booked" or "there are no more flights on 10.8.2005 from X to Y" are derived from "person P is booked for seat 5C of flight LH123 on 10.8.2005" under certain conditions. Note that there can be multiple derivations for a derived event (for the second one above, also e.g. "flight AF678 on 10.8.2005 is cancelled").

Raising and Derivation of Events. The task of becoming aware of implicit events is not the task of the rule execution, but of application-level reasoning, based on the application ontology. Thus, there are *derivation rules* also for events (that can be seen as event-condition-action rules where the action consists of raising an event); see Section 3.1.5.

Localization of Events. Orthogonal to being derived or not, application-level atomic events can be associated with a certain node or can describe happenings on the Web-wide level:

- local events: these happen locally at the node. E.g., data model events are in most cases used locally (by triggers, which then can raise higher-level events or trigger a remote action)
- remote events: From the point of view of a rule, a *remote* event is an event that is local (and can be localized) at another node, e.g., "if Amazon offers a new book on X". Here, event detection can be done e.g. by monitoring the node (*continuous querying*) or by a *publish-subscribe-service.*
- global events: global events happen "somewhere in the Web", e.g., "a new book on the Semantic Web is announced", or "election of a new chancellor of Germany". *Global* events are (mostly) application-level events where it is not explicitly specified where they actually occur.

In these cases, event detection is even more complicated since it must also be searched and derived *where* and *how* the event can be detected. Rules using global events require appropriate communication and notification mechanisms by the Semantic Web infrastructure (that can in turn also be based on ECA rules). For dealing with global (not located or locatable at a certain node) implicit (derived), the Semantic Web must provide *event broker* functionality.

Temporal Delay and Event Wrapping. Event brokering leads to the effect that the time-points of actual events and the event detection may differ. The actual infrastructure will probably also use messaging functionality, i.e., not raising an event, but packing it in a message or event "I became aware that ...".

Example 3.1 *Consider a customer C who wants to buy a Christmas tree, C can state the following rules:*

- *"if some X announces to sell Christmas trees, go there". The rule is correct, but not "complete": Probably, C will not become aware of the event of announcing, so he will never get a Christmas tree.*
- *"if I become aware [by a message] that some X announced to sell Christmas trees, go there". This rule is more complete, but, if C becomes aware too late (e.g., after New Year) he will also go there.*
- *The correct rule is thus "if I become aware [by a message] [before Christmas] that some X announced to sell Christmas trees, go there".*
- *Nevertheless, the real-world formulation of the rule will be of the style "if some X sells Christmas trees, I should go there" – which is formally based on an event that X sells a tree to some Y (which will eventually happen, but C will most probably not be informed about it).*

Thus, in a future stage of Semantic Web rules, the interpretation of rules should provide a reasoning-based rewriting of rules to get their intended semantics.

Composite Events. Composite events are subject of heterogeneity in that there are multiple formalisms and languages for describing them. As already mentioned, most of them use *event algebras*. The target framework for the Semantic Web must support this heterogeneity.

3.1.5 Types of Rules

In our ECA-based approach, the behavior of domains *described* in an axiomatic way by (OWL) ontologies is specified and implemented by ECA rules. In a first step, these descriptions have to allow to run applications by ECA rules. In the next step, the ontology has to be extended to preconditions and effects as a base for *reasoning*.

There are several types of rules that are used for actually specifying the ontology and the behavior of an application:

- Rules that axiomatize the *ontology*, i.e., mandatory relationships between actions, objects, and events that are inherent to the domain. The correctness of the rules must be proven against the ontology.
- Rules that specify a given *application* on this domain, e.g., business rules. Changing such rules result in a different behavior of the application.

ECA Rules. From the external user's point of view, *ECA-Business Rules* specify the actual behavior: "when something happens and some conditions are satisfied, something has to be done". Here, events and actions refer to a very high and abstract level of the ontology.

- Such rules are "actual", user-defined ECA rules since they trigger an action upon an event "to keep the application running". Such rules exist on different abstraction levels and granularity, designed to the notions of the application domain. Changing them changes the behavior of the application.
- Internally, such rules are also used for implementing mechanisms for detection of derived and composite events on the respective level.

ECE Event Derivation Rules: Providing High-Level Events. For implementing high-level rules, it is necessary that these high-level events are provided somehow: They must be *derived.*

- horizontal ECE rules: Here, the event is derived from another high-level event under certain conditions, e.g., *"when a booking for a flight is done, and this is the last seat, then the plane is completely booked"*. The rule is an E-C-E (event-condition-event) rule, e.g., the "action" consists in deriving/raising an event. The events are logically related and inherent in terms of the application. Changing such rules would invalidate the application wrt. its ontology.
- upward vertical ECE rules: an abstract event is derived from changes in the underlying database, e.g., *"when the arrival time in a database of a flight of today is changed, this is actually a delayed flight"*. The rule is again an E-C-E (event-condition-event) rule. The events are not logically related and inherent in terms of the application, but are related due to the physical implementation of the application (i.e., since an explicit message "flight F is delayed" is missing, and only visible due to a modification of the database". Changing such rules would invalidate the application wrt. its ontology.

  As another example, "professor_hired($object, $subject)" is (locally) derived at a node from an insertion of a fact into an SQL, XML, or RDF database.

These rules correspond to the *bottom-up* semantics of derivation rules: Given the body, do the head. While "classical" ECA rules are *active* rules, the above enumeration presented several kinds of *derivation* rules. The main difference of these wrt. classical *derivation* is that the latter define continuously existing *views* and are used for *querying.* In contrast, the *event derivation rules* "fire" only once when an event is detected and another event must be raised. Thus, these *ECE* rules are more similar to ECA rules than to derivation rules.
The derivation of events by such rules can be done similar to views:

- bottom-up style: they can be "materialized" by raising them explicitly when (and where) they occur (even if probably nobody is actually interested in them), or
- top-down style: when an application uses a derived event, it runs the rule locally.

High-level events can also be raised as side-effects of high-level actions, see below. When designing rules, it must be cared that such effects do not cause any behavior twice.

ACA/ACE Rules: Talking about High-Level Actions. High-level actions like "(at a travel agency) book a travel by plane from Hannover to Lisbon" cannot be executed directly, but there must be another rule that says how this is implemented (by searching for connections, possibly via Madrid). Such rules are *reduction* rules that reduce an abstract action to actions on a lower level.

On the other hand, there may be another business rule that should be executed whenever somebody does a plane travel from Germany to Portugal, putting this person on a list for sending them advertisements about (questionable) tax saving tricks by investing in resorts in the Algarve. The latter rule should not be defined on the basis of "if there are bookings for a person via some places that lead from Germany to Portugal" (which would e.g. also fire if a Tyrolian flies from Innsbruck to Munich and then to Lisbon; the german tax tricks do probably not apply to him), but could –most declaratively– directly use the *abstract action* "book a travel by plane from a german airport to a portuguese airport" for firing the rule.

Thus, there are rules that regard (abstract) actions as events – or in the above case more exactly, use the event of *committing an abstract action.* Such rules can be expressed based

on transactional events, or on messages ("if we get the message that such an abstract action should be executed"), but in both cases this blurs the declarativity that the *action* actually is the reason to react.
Thus, there are several kinds of ACA rules:

- horizontal reduction ACA rules, e.g., *"the action of transferring 200E from account A to B is implemented by debiting 200E from account A and deposing 200E on account B"*. This rule is kind of an A-C-A rule that explains a composite action by its components, both still in terms of the application domain.
- vertical reduction ACA rules, e.g., *"the action of debiting 200E from account A is realized by reading the account value, adding 200 and writing it"*. This rule is also a kind of an A-C-A rule that reduces a composite action into its components on a lower level.
- horizontal non-reduction ACA rules see an action (that has to be executed for itself) as an event that should trigger another action, known also as *rule chaining*.

  This kind of ACA rules is more directly related to ECA rules. Changing such rules would not invalidate the application wrt. the ontology, but just change its behavior.

The above reduction ACA rules correspond to SQL's `INSTEAD` triggers: in SQL, `INSTEAD`-triggers are used for specifying what updates should actually be done instead of inserting something into a view (which is not possible), whereas here, simpler actions are specified instead of an abstract one. Such rules are closely related to the *top-down* semantics of derivation rules: To obtain the head, realize the body (cf. Transaction Logic Programming [15]). These rules describe actions that are logically related and inherent in terms of the application. Some of these rules are inherent to the ontology of the underlying domain, others specify only the behavior of a given application (including e.g. policies that are not inherent to the domain).

Since high-level events can also be seen as observations, it is also reasonable to raise them when an appropriate action is executed. In most cases, this amounts to a simple mapping from high-level actions to raise high-level events: the action "hire_professor($object, $subject)" directly raises the event "professor_hired($object, $subject)" at the same node (and is internally executed by inserting a fact into an SQL, XML, or RDF database; see downward ACA vertical rules above).
The execution of both, ACA and ACE rules, must usually be located at the node where the action is executed (which makes completely sense because ACA is actually the algorithm to execute an action, and ACE is the communication of its effect on a high level).

Low-Level Rules. The base is provided by update actions on the database level (to which all abtract actions must eventually be reduced to actually change the state of any node) and low-level ECA rules, e.g., database triggers for referential integrity or bookkeeping.

Here, neither the event nor the action is part of the application ontology, but both exist and are related only due to the physical implementation of the application. Such rules guarantee –together with the RDF views on the database– the integrity of the model; thus, when verifying a process they also have to be taken into account.

Summary and Interference. The resulting information flow between events and actions is depicted in Figure 3.4.

Example 3.2 (Actions and Events) *Consider the following scenario: A –rule-driven– review process leads to the acceptance of a paper. Here, "acceptance of a paper" is an action*

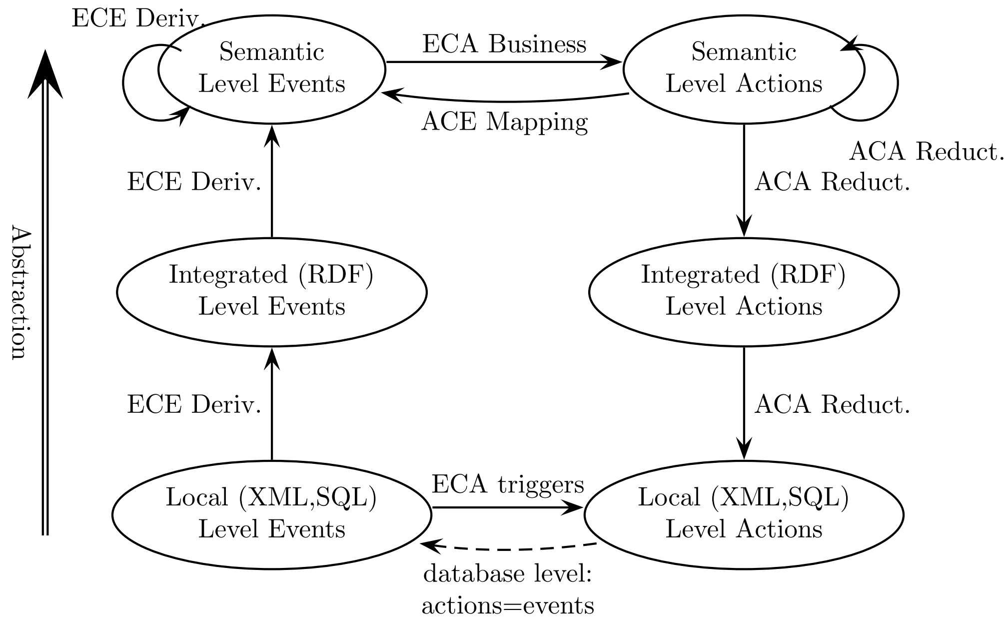


Figure 3.4: Interference of Events, Actions, and Rules

which is also an event on the Web level – "the paper P of A has been accepted for conference C". This event is communicated inside the program committee by mail (push *communication). An internal rule of the conference of the form "when a paper is accepted, send a message to the author and list it on the conference Web page". By the latter, the event of "the paper P of A has been accepted" becomes actually accessible for the public. Communication in the Semantic Web then should lead to firing other business rules, e.g., at the DBLP publications server and at citeseer, where lots of "business rules" (later) react upon.*

Such rules in the Semantic Web are not formulated on the level of messaging, but assume the notification about events as given (which is, on a lower level, done by messaging).

The author of an accepted paper probably has a rule "when a paper is accepted (event), then book a travel to the conference". This action is submitted as a message "Person P wants to book a travel from X to Y on date D" to a travel agency. The travel agency reacts on incoming messages by searching for connections and book an available one, possibly by flights AA123 from X to Z and BB456 from Z to Y (horizontal rule for decomposing an action into its constituents). These actual bookings of flights are then submitted to the airline, seats are assigned, and the booking actually takes place by modifications of the database content (vertical rules). Assume that this booking reserves the last seat in AA123.

On the other hand, there are several other rules that should fire in this process, e.g., a rule that removes all completely booked flights from some list, and that raises the price for all flights between two destinations M and N in case that more than 50% of the total capacity on this connection for that day is booked.

Both can be done by vertical rules, reacting on database events for booking actual seats, or on the higher level, e.g. "on any booking between M and N (event), check all flights between two destinations M and N, and in case that more than 50% of the total capacity on this connection

for that day is booked, raise the price for the remaining ones by 10E". In the latter case, the booking action is immediately also seen as an event "somebody books ...".

Moreover, there can be a business rule "whenever a person P books a travel from country C to country D, do ...". If X is located in C and Y is located in D, but Z is located in a different country, then, this event cannot be detected from the actual, independent bookings of the individual flights. Instead, the action "book a travel from X to Y on date D for person P" should be considered as a high-level event for immediately firing appropriate rules.

3.2 Simple ECA Rules: Data Model Triggers

A simple form of active rules that is often provided by database systems, are *triggers*. As shown in Figure 3.4, these triggers on the database level form the lowest level of rules. Reacting directly to changes in the database, they provide the basic level of behavior. Triggers are simple rules on the *(database) programming language and data structure level*. They are associated with the logical level (i.e., not referring the the implementation of the logical data model, but acting on its notions). They follow a simple ECA pattern where the conditions are given in the database query language and the action component is given in a simple, operational programming language. In SQL, triggers are of the form

ON database-update WHEN condition BEGIN pl/sql-fragment END .

In the Semantic Web, this base level is assumed to be in XML or RDF format. While the SQL triggers in relational databases are only able to react on changes of a given tuple or an attribute of a tuple, the XML and RDF models call for more expressive event specifications according to the (tree or graph) structure.

3.2.1 Triggers on XML Data

Work on triggers for XQuery has e.g. been described in [12] with *Active XQuery* (using the same syntax and switches as SQL, with XQuery in the action component) and in [5, 57], emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases. The following proposal refines our previous one developed in [3]:

- `ON {DELETE|INSERT|UPDATE} OF` *xsl-pattern*: if a node matching the *xsl-pattern* is deleted/inserted/updated,
- `ON MODIFICATION OF` *xsl-pattern*: if anything in the subtree rooted in a node matching the *xsl-pattern* is modified,
- `ON INSERT INTO` *xsl-pattern*: if a node is inserted (directly) into a node matching the *xsl-pattern*,
- `ON {DELETE|INSERT|UPDATE} [SIBLING [IMMEDIATELY]] {BEFORE|AFTER}` *xsl-pattern*: if a node (optionally: only sibling nodes) is modified (immediately) before or after a node matching the *xsl-pattern*.

All triggers should make relevant values accessible, e.g., `OLD AS ...` and `NEW AS ...` (like in SQL), both referencing the node to which the event happened, additionally `INSERTED AS`, `DELETED AS` referencing the inserted or deleted node.

Similar to the SQL `STATEMENT` and `ROW` triggers, the granularity has to be specified for each trigger:

- `FOR EACH STATEMENT` (as in SQL),
- `FOR EACH NODE`: for each node in the *xsl-pattern*, the rule is triggered only at most once (cumulative, if the node is actually concerned by several matching events) per transaction,
- `FOR EACH MODIFICATION`: each individual modification (possibly for some nodes in the *xsl-pattern* more than one) triggers the rule.

The implementation of such triggers in XML repositories can e.g. be based on the *DOM Level 2/3 Events* or on the triggers of relational storage of XML data. Events/triggers on this logical level are local (and internal) to the database that provides an RDF view to the outside. Usually, the actions are local updates of the database (that then effect the RDF view indirectly), or they raise events on the RDF level (it is see also below), but it is also possible to send XUpdate or SOAP messages to other nodes, or to state remote XML updates explicitly:

```
ON INSERT OF department/professor
let $prof:= :NEW/@rdf-uri, $dept:= :NEW/parent::department/@rdf-uri
RAISE RDF_EVENT(INSERT OF has_professor OF department)
  with $subject:= $dept, $property:=has_professor, $object:=$prof;
RAISE RDF_EVENT(CREATE OF professor)
  with $class=professor, $resource:=$prof;
```

XML: Local and Global Rules in the "Conventional" XML Web. The above events occur always local in a node and can be detected at this node.

Rules on the XML level of the Web can either be local to a certain node, or they can include Web data, e.g., reacting on events on views that include remote data, or raising actions on the Web. For that reason, we call them *Web Level Triggers* (note that these can already be applied to the conventional non-semantic Web; whereas for integration reasons in the *Semantic Web*, the level of RDF events is preferable).

Actual rules on this level usually are not only based on atomic data-level events, but use own event languages that are based on a set of atomic events (that are not necessarily just simple update operations) and that usually also allow for composite events. Their event detection mechanism is not necessarily located in the database, but can be based on the above triggers.

Such rules require knowledge of the actual XML schema of the corresponding nodes. Provided a mapping between rules on the XML level and those of the RDF view level, the implementation can (more efficiently) be kept on the XML level, whereas reasoning about their behavior can be lifted to the Semantic Web level. A Semantic Web framework should also support this kind of rules.

From the *Semantic* Web point of view, events on the XML level should usually not be communicated to other nodes (except very close coupling with nodes using the same schema); instead semantic events should be derived from them.

3.2.2 Triggers on RDF Data

RDF triples describe properties of a resource. In contrast to XML, there is no subtree structure (which makes it impossible to express "deep" modifications in a simple event), but there is metadata. A proposal for RDF events can be found in RDFTL [57, 58]. The following proposal refines our previous one developed in [3]:

- `ON {INSERT|UPDATE|DELETE} OF` *property* `[OF` *class*`]` is raised if a property is added to/updated/deleted from a resource (optionally: of the specified class).

- ON {CREATE|UPDATE|DELETE} OF *class* is raised if a resource of a given class is created, updated or deleted.

On the RDF/RDFS level, also metadata changes are events:

- ON NEW CLASS is raised if a new class is introduced,
- ON NEW PROPERTY [OF CLASS *class*] is raised, if a new property (optionally: to a specified class) is introduced.

Besides the OLD and NEW values mentioned for XML, these events should consider as arguments (to bind variables) Subject, Property, Object, Class, Resource, referring to the modified items (as URIs), respectively. Trigger granularity is FOR EACH STATEMENT or FOR EACH TRIPLE.

Application-level events (that must be characterized appropriately in the application ontology) can then be raised by such rules, e.g.,

```
ON INSERT OF has_professor OF department
  % (comes with parameters $subject=dept, $property:=has_professor,
  %    and $object=prof)
  % $university is a constant defined in the (local) database
RAISE EVENT (professor_hired($object, $subject, $university))
```

which is then actually an event (e.g., professor_hired(*prof*, *dept*, *univ*)) of the application ontology on which a "business rule" of a publisher could react that says, if a new professor is hired at a university, then the appropriate list of textbooks should be sent to him. Note that in the above trigger, this event is only raised – the complete issues of communicating it and detecting it by the node that actually processes the business rule have to be dealt with separately.

Semantic level. In rules on the semantic, OWL level, the events, conditions and actions refer to the domain ontology. Concerning detection of such events, there are some differences: plain RDF events can be detected on the data level. Since OWL is not the immediate data model, events on the OWL level cannot straightforwardly be detected from any (RDF) database event, but in general require OWL reasoning. Even more, often the local knowledge of the node is not sufficient, but in general, OWL events refer to the distributed scenario.

We give here just an example to complete the "upward" transmission of events and show the difference between the above, mainly syntactical, levels and the semantic level:

```
ON (professor_hired($prof, $dept, $univ))
WHEN $Books := select relevant books for people at this dept
BEGIN do something END
```

Here, the difference between *actions* and *events* shows up: an event is a visible, possibly indirect or derived, consequence of an action. E.g., the action is to "debit 200E from Alice's bank account", and visible events are "a change of Alice's bank account" (that is immediately detectable from the update operation), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update).

More complex rules also use composite events and queries against the Web. Composite events in general consist of subevents at different locations. Additionally, higher-level events are not necessarily associated with a given database, and are in general not explicitly raised. In the above example, both was simple: the source explicitly raised professor_hired($object, $subject, $university), and the publisher can e.g. register at all universities to be notified about such events. In general, events like "when a publication p becomes known that deals with ...")

cannot be detected in this simple way, bust must be derived and obtained from other, more general information. Here, Semantic Web reasoning comes heavily into play even for detecting atomic events "somewhere in the Web". Here, the timepoints of actual events and the event detection may differ.

3.2.3 Triggers vs. ECA Rules

As shown in Figure 3.4, triggers on the database level form the lowest level of rules. They deal with data model instead of the application level. Thus, their "home" is inside the database, using notions of the database model, and their implementation often event depends an the availability of information from the physical level of the database. Especially, their "events" coincide directly with the update operations of the database, which are also the *actions* on that level. Triggers are not necessarily subject of the modularization of our model. In contrast, often triggers, although following the ECA paradigm, are only very restricted. They are usually closely intertwined with the database (e.g., in relational databases), although very efficient external implementations exist, cf. TriggerMan [40].
In case that triggers are implemented external to a database, the data item where the triggering event occurs must usually be identifiable and re-findable:

- SQL: OLD and NEW are the modified tuples; all related data can be identified externally by foreign keys. Furthermore, usually, the ROWID is used that identifies a tuple internally.
- XML: OLD and NEW should at least allow to access the subtrees, but also parent or ancestor nodes may be of interest. For this, an internal identifier must be accessible. Note that there may also be triggers that do a modification inside the database relative to the modified node, e.g., in the subtree.
- RDF: OLD and NEW are the current nodes. Since they in general have a URI, they can be identified without any problem.

These triggers are usually defined in a homogeneous way on the programming language level (i.e., data model events of the data model's update language, queries in the data model's query language, and actions are given in as data model updates or as a program segment in a programming language that includes the data model's query language). On the higher level, ECA rules make use of more abstract languages.

3.3 ECA Language Structure

After having discussed and analyzed the requirements for specifying and implementing behavior in the Semantic Web by active, ECA-style rules, we develop now the structure of such rules and the required languages.

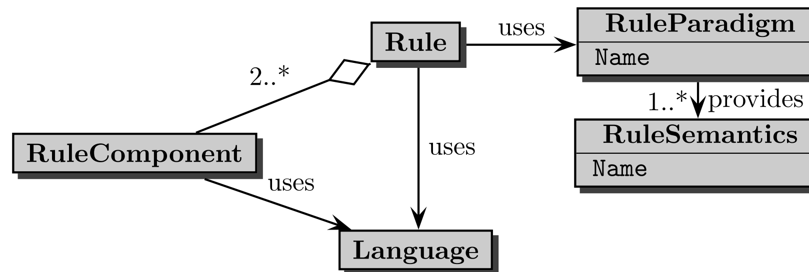
3.3.1 Language Heterogeneity and Structure: Rules, Rule Components and Languages

Rules in general consist of several components. For instance, deductive rules like in Prolog/Datalog consist of a *rule head* and a *rule body*; similar e.g. for F-Logic or Transaction Logic rules. These languages are *rule-based languages* – their head and body are both expressions in this language (but note that e.g. negation may in most cases only occur in rule bodies). In the ECA

paradigm, rules are not homogeneous: they consist of an *event component*, a *condition component* (that together roughly correspond to the "body" since they describe the situations where the rule is applicable), and an *action component* (that roughly corresponds to the "head") – and all these components use different languages. For example, for database triggers, events are database events like "on update" (that are raised by update operations), where-clause conditions are expressed in SQL, and actions are SQL programs or updates (that again may raise events). Another example is the language XChange discussed in Section 2, where the event component consists of an event query, the condition component is a test by evaluating an Xcerpt query, and the action component may contain update actions, transactions, or raising of an event. Yet another example can be found in this report in Appendix A, for ruleCore, where the (different) languages for each component are described. The ECA languages mentioned in the introduction ([13, 12, 5, 58]) also use different languages in each of the components. In these cases, the event, condition, and action language are usually closely related, but this is not necessarily always the case. The more complex a scenario is, the more specialized are the used component languages.
Thus, the semantics of a rule is determined by two constituents:

- the rule semantics or "rule paradigm" that characterizes the interplay between the components, and the
- language(s) used in its components.

E.g., *deductive rules* have several common semantics, either top-down, bottom-up as fixpoints, or well-founded or stable semantics, independent what underlying language (first-order logic, F-Logic, Transaction Logic etc.) is used. In the same way, ECA rules have a fixed semantics, independent what languages are used in the E, C, and A components. An important common feature here, that has been well discussed already for XChange, is that the communication both for derivation rules and for ECA rules is done by *logical variables*; we will discuss this in detail in Section 3.4.



Constraints:
RuleParadigm determines number of RuleComponents
RuleComponents ordered or named; using appropriate languages

Figure 3.5: Rules, Rule Components and Languages

An XML markup of rules must cover the *structure* of rules and rule component languages. Here, the RuleML language [63] provides *general* guidelines that must then be specialized for each paradigm.

In the following, we will investigate general ECA rules. The analysis of the languages will

be continued in two aspects: semantics/ontology and syntax (i.e., algebraic, variables etc.).

3.3.2 Components and Languages of ECA Rules

In usual Active Databases in the 1990s, an ECA language consisted of an event language, a condition language, and an action language. For use in the Semantic Web, the ECA concept needs to be more flexible and adapted to the "global" environment of a world-wide living organism where nodes "speaking different languages" should be able to interoperate. So, different "local" languages, for expressing events, queries and conditions, and actions have to be integrated in a common framework.

The target of the development and definition of languages for (ECA) rules and their components should be a semantic approach, i.e., based on an (extendible) ontology for these notions that allows for *interoperability* and also turns the instances of these concepts into objects of the Semantic Web itself. The upper level of this ontology is shown as an UML diagram in Figure 3.7, which will be explained below.

In contrast to previous ECA languages from the database area, we aim at a more succinct, conceptual separation between the event, condition, and action components, which are (i) possibly given in separate languages, and (ii) possibly evaluated/executed in different places. Each of the components is described in an appropriate language, and ECA rules can use and combine such languages flexibly.

Analysis of Rule Components. A basic form of ECA/active rules are the well-known *database triggers*, e.g., as already shown above, in SQL, of the form

ON *database-update* WHEN *condition* BEGIN *pl/sql-fragment* END.

For them, *condition* can only use very restricted information about the immediate *database update*. In case that an action should only be executed under certain conditions which involve a (local) database query, this is done in a procedural way in the *pl/sql-fragment*. This has the drawback of not being declarative: reasoning about the actual effects would require to analyze the program code of the *pl/sql-fragment*. Additionally, in the distributed environment of the Web, the query is probably (i) not local, and (ii) heterogeneous in the language – queries against different nodes may be expressed in different languages. For our general framework, we prefer a *declarative* approach with a *clean, declarative* design as a "Normal Form": Detecting just the dynamic part of a situation (event), then check *if* something has to be done by probably obtaining additional information by a query and then evaluating a *boolean* test, and, if "yes", then actually *do* something – as shown in Figure 3.6.

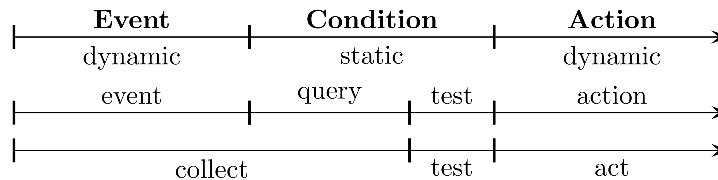


Figure 3.6: Components and Phases of Evaluating an ECA Rule

With this further separation of tasks, we obtain the following structure:

- every rule uses an event language, one or more query languages, a test language, and one or more action languages for the respective components (for that, we allow several action components in different languages that have *all* to be executed,
- each of these languages and their constructs are described by metadata and an ontology, e.g., associating them with a processor,
- there is a well-defined *interface* for communication between the E, Q&T, and A components by variables (e.g., bound to XML or RDF fragments).

This model can be readily extended by adding a fifth optional component – the post-condition (another *test*) – resulting in a variation usually called ECAP rules. In most cases, this post-condition can be omitted by allowing the action language to test for conditions inside the action component. But it may have particular relevance when considered together with transactional rules, and for reasoning about the effects of sets of rules.

For applying such rules in the Semantic Web, a uniform handling of the event, query, test, and action sublanguages is required. For this, rules, their components, and the languages must be objects of the Semantic Web, i.e., described in XML or RDF/OWL in a generic *rule ontology* that contains all required information as shown in the UML model in Figure 3.7.

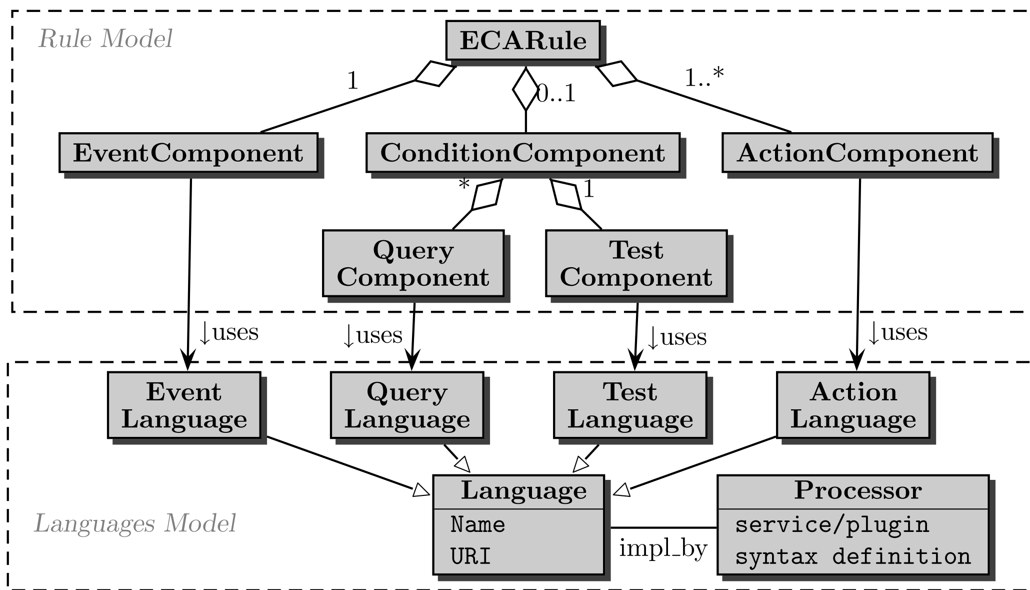


Figure 3.7: ECA Rule Components and Corresponding Languages

Markup Proposal: ECA-ML

The model is accompanied by an XML ECA rule (markup) language, ECA-ML. The relationship between the rule components and languages is provided by identifying the languages with namespaces (from the RDF point of view: resources), which in turn are associated with information about the specific language (e.g, an XML Schema, an ontology of its constructs, a

URL where an interpreter is available). The latter issues are discussed in Section 3.7; here we investigate the languages and the markup itself.

For an XML representation of ECA rules as shown in Figure 3.7, we propose the following (basic) markup (ECA-ML):

```
<!ELEMENT rule (%variable-decl,event,query*,test?,action+)>
  <!-- %variable-decl is not yet specified in detail here-->

<eca:rule rule-specific attributes>
  rule-specific content, e.g., declaration of logical variables
  <eca:event identification of the language >
    event specification, probably binding variables; see Section 3.5.1
  </eca:event>
  <eca:query identification of the language >     <!-- there may be several queries -->
    query specification;   using variables, binding others; see Section 3.5.2
  </eca:query>
  <eca:test identification of the language >
    condition specification, using variables; see Section 3.5.3
  </eca:test>
  <eca:action identification of the language >     <!-- there may be several actions -->
    action specification, using variables, probably binding local ones; see Section 3.5.5
  </eca:action>
</eca:rule>
```

The actual languages (and appropriate services etc.) are identified by namespaces and their declarations in the <eca:...> elements (see Section 3.6 for component languages and Example 3.14 later).

A similar markup for ECA rules (without separating the query and test components) has been used in [13] with *fixed* languages (a basic language for atomic events on XML data, XQuery as query+test language and SOAP in the action component). This fixed approach falls short wrt. the language heterogeneity, and especially the use and integration of languages for composite events. The same structure is also followed by the XChange transaction rules above, there again without any intention to deal with heterogeneity of languages: fixed languages are used for specifying the event, condition (without separating query/test), and action component. In contrast, here we generalise the approach to allow for using arbitrary languages. Thus, these other proposals are just possible configurations. Our approach even allows to mix components of both these proposals.

3.3.3 Opaque Rules

Moreover, there are the existing trigger-style languages that handle specific, simple database events, simple conditions and actions, with their own syntax as discussed in Section 3.2 above. Since these triggers work on the logical level and are in general (very efficiently) implemented based directly on the physical database level, they are not necessarily marked up in ECA-ML. Often, they are even not subject to the “Semantic Web”, since they are just used to locally *implement* something that is *specified* in a completely different way (e.g., integrity constraints), or for raising RDF-level events based on an SQL or XML storage. In our ontology, we embed this as *opaque* rules as shown in Figure 3.8.
The ECA-ML language provides the following XML markup for opaque rules:

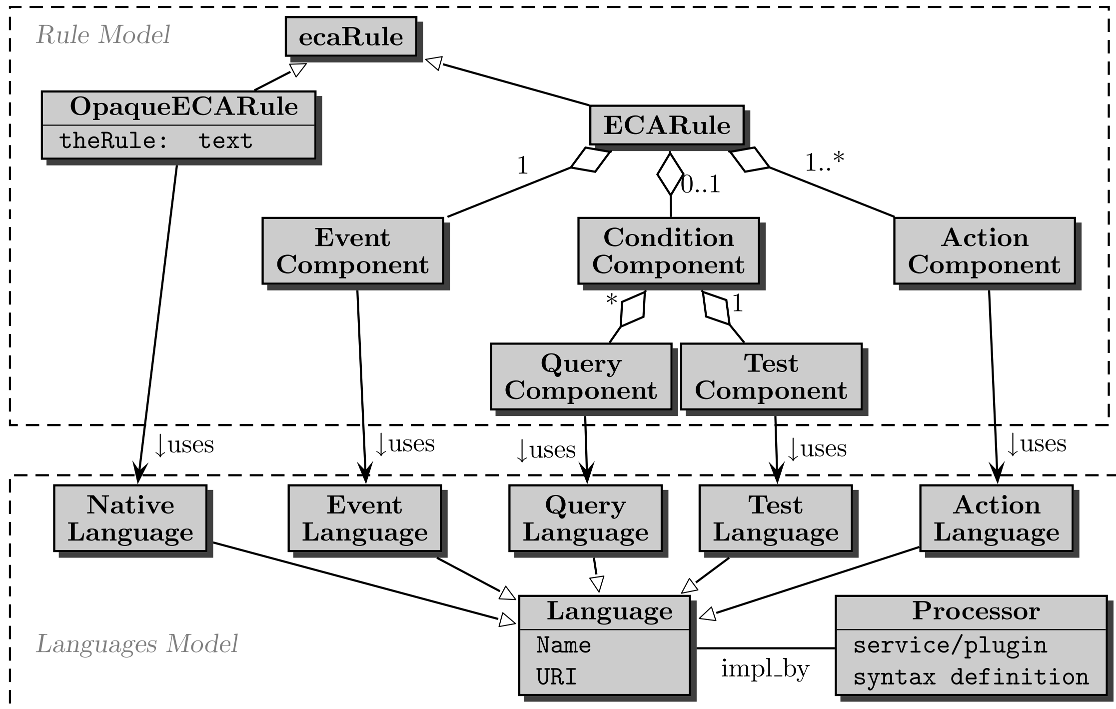


Figure 3.8: ECA Rule Components and Corresponding Languages II

- eca:opaque with text content (program code of some rule language) and attributes lang (text) and ref (URI where an interpreter is found, similar to the namespace). The attributes could be alternative if lang is designed similar to XML's NOTATION concept that associated resources with a program URI.

  Non-normative example:

  <**eca:rule**>
  <**eca:opaque** name="SQL trigger" ref= "*uri of the trigger language*">
  ON *database-update* WHEN *condition* BEGIN *action* END
  <**/eca:opaque**>
  <**/eca:rule**>

- eca:opaque with element content of a language that provides an opaque syntax itself; then, there is only a namespace declaration for this language:

  Non-normative example:

  <**eca:rule**>
  <**eca:opaque** xmlns:*foo*="*uri of the trigger language*">
  <foo:trigger>
  ON *database-update* WHEN *condition* BEGIN *action* END
  </foo:trigger>
  <**/eca:opaque**>

</**eca:rule**>

Note that <foo:trigger> is an element of the markup wrapper language for the native language.

This point of view also allows to have rules in any other ECA markup language and just to "forward" them to "their" service.

Since opaque rules are ontologically "atomic" objects, their event, condition, and action components cannot be accessed by Semantic Web concepts.

Note that there are canonic mappings between such triggers and their components and the general ECA ontology.

3.3.4 Hierarchical Structure of Languages

The approach defines a hierarchical structure of language families (wrt. the embedding of language expressions) which relates the different kinds of ontologies (application-dependent and application-independent) and their components as already described above in Section 3.1.3 and Figure 3.1 to the languages of rules and rule components as shown in Figure 3.9 [here directly associating ontologies with the corresponding languages over the same alphabet]: As described until now, there is an ECA language (that is already described by the above markup), and there are (heterogeneous) event, query, test, and action languages. Rules will combine one (or more) language of each of the families. In general, each such language consists of its own, application-independent syntax and semantics (e.g., event algebras, query languages, boolean tests, process algebras or programming languages) that is then applied to a domain (e.g. travelling, banking, universities, etc.). The domain ontologies define the static and dynamic notions of the application domain, i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of train schedule changes, actions of reserving tickets, ...). Additionally, there are domain-independent languages that provide primitives (with arguments), like general communication, e.g. received_message(M) (where M in turn contains domain-specific content).

In the next section, we discuss common aspects of the languages on the "middle" level (that immediately lead to the tree-style markup of the respective components – thus, here the XML markup is straightforward). Communication issues between the rule components are discussed in Section 3.4 before we come to the semantics of rule execution in Section 3.5. Samples of component languages will be discussed in Section 3.6; a short account on domain languages including events and actions has been given in Section 3.1.3.

3.3.5 Common Structure and Aspects of E, C, T and A Sublanguages

The four types of rule components use corresponding types of languages that share the same algebraic language structure, although dealing with different notions:

- event languages: every expression gives a description of a (possibly composite) event. Expressions are built by composers of an event algebra, and the leaves here are atomic events of the underlying application domain or an application-independent domain;
- query languages: expressions of an algebraic query language, embedding literals from the domains;
- test languages: they are in fact formulas of some logic over literals of that logic in the underlying domains (that determine the predicate and function symbols, or class symbols etc., depending on the logic);

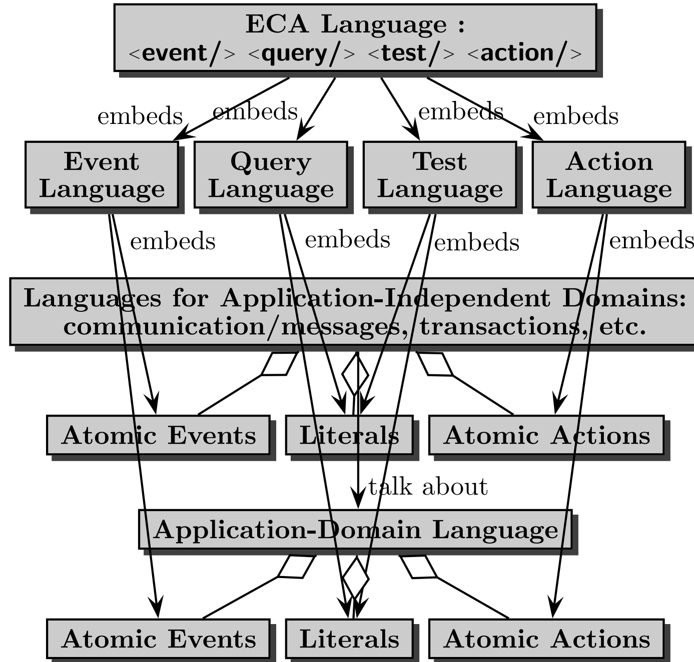


Figure 3.9: Hierarchy of Languages

- action languages: every expression describes an action. Here, algebraic languages (like process algebras) or "classical" programming languages (that nevertheless consist of expressions) can be used. Again, the atomic items are actions of the underlying domains.

Algebraic Languages.

As shown in Figure 3.10, all components have in common that the component languages consist of an algebraic language defining a set of *composers*, and embedding *atomic* elements (events, literals, actions) that are contributed by the *domain languages*. Expressions of the language are then (i) atomic expressions, or (ii) composite expressions recursively obtained by applying composers to expressions. Due to their structure, these languages are called *algebraic languages*, e.g. used in *event algebras*, *algebraic query languages*, and *process algebras*. Each composer has a given *cardinality* that denotes the number of expressions (of the same type of language, e.g., events) it can compose, and (optionally) a sequence of parameters (that come from another ontology, e.g., time intervals) that determines its *arity* (see Figures 3.10 and 3.11).

For instance, "E_1 followed_by E_2 within t" is a binary composer to recognize the occurrence of two events (atomic or not) in a particular order within a time interval, where t is a parameter. Event languages define different sets of composers, such as XChange for its composite event queries (as shown in Section 2.2.3.3), the ruleCore detectors as shown in Appendix A.2.1.2, or the SNOOP event algebra of [25]. Similar composers are used in process algebras, or also –but in general syntactically covered– in algebra-based query languages. The boolean algebra with its composers is well-known.

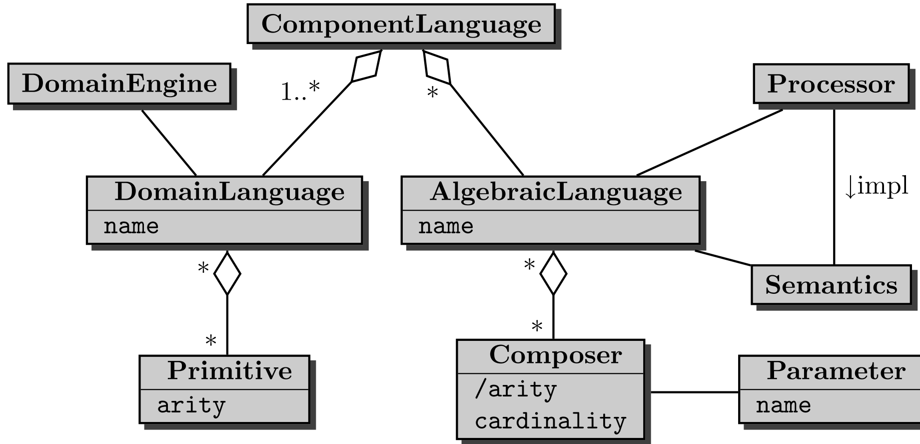


Figure 3.10: Notions of an Algebraic Language

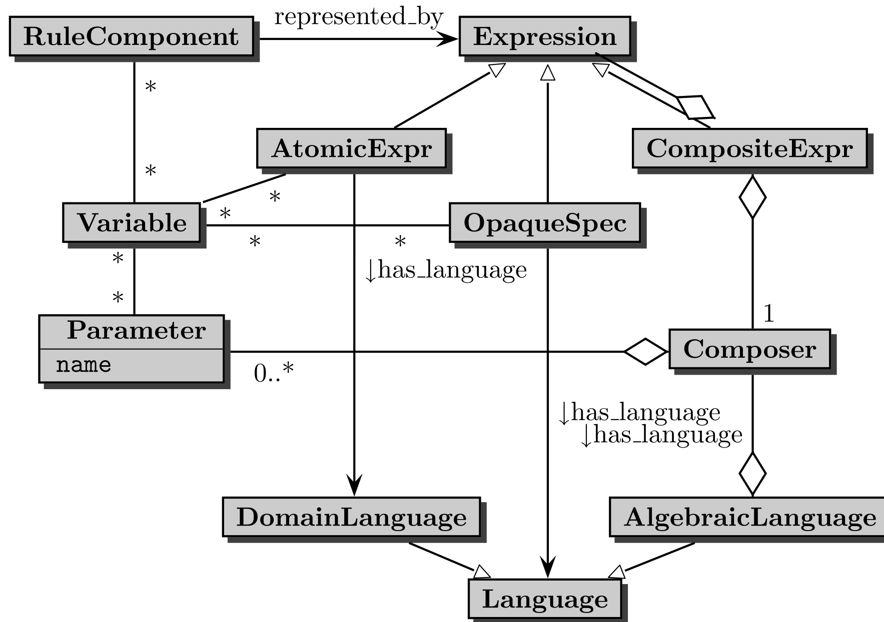


Figure 3.11: Syntactical Structure of Expressions of an Algebraic Language

Semantics of Algebraic Languages.

Every algebra expression is assigned a semantics, i.e., from evaluating it (in case of queries: in a given state). Starting with the semantics of atomic expressions, the semantics of composite expressions is determined by the composer. The semantics of the different types of algebraic languages are as follows (see also Section 3.6):

- event languages: in most cases, the event instance(or sequence) that matches the given expression pattern,
- query languages: a query result, e.g., a relation or an XML structure,
- test languages: a truth value of a logic (i.e., for classical logics, true or false)
- action languages: the formal semantics of terms of e.g. process algebras are denotational or operational semantics as state transformers; nevertheless, here we are only interested in their side effects on the underlying data.

Composition of Algebraic Languages.

For each type of such algebraic languages (i.e., event, query, test, and action languages), the expressions define and combine entities of the same kind, i.e., again, events, queries, tests, or actions, possibly with appropriate parameters and logical variables (see below) and using events, literals, or actions from the respective part of the domain language. Thus, from the ontology point of view, entities of the same kind described in different languages can be combined (e.g., a conjunction of a (sub)formula in first-order logic with one in description logic, or a sequence (or, more generic, any binary composer of any event language) of an event specified in language EL_1 and one specified in EL_2).

The global language concepts and the markup will support this, and we will also explain how evaluation also smoothly crosses these language borders. Note that Figure 3.11 does not associate the whole rule components with a language, but each *expression* is associated to a language. Sections 3.4 and 3.5 show how such cross-language functionality is provided.

Tree Markup of Algebraic Languages.

Thus, language expressions are in fact trees which are marked up accordingly. The markup elements are provided by the definition of the individual languages, "residing" in and distinguished by the appropriate namespaces. As described above, it is also possible to nest composers and expressions from different languages of the same kind, distinguishing them in the markup by the namespaces they use. Thus, languages are in fact not only associated once on the *rule component* level, but this can also be done on the *expression* level.

Opaque Rule Components. Analogous to *opaque* rules, we allow for opaque rule components, e.g., conditions that are not expressed in a markup language, but in XQuery, or actions expressed in Java or Perl. In such cases, again <eca:opaque> elements are used that refer to a URI for that language.

Language Binding. As shown in Figure 3.11, every expression (i.e., the rule components and their subexpressions) can be associated with a language: an expression is either

- an atomic one (atomic event, literal, action) that belongs to a domain language (either application-dependent or application-independent), or
- an opaque one that is a code fragment (not in XML markup) of some event/query/logic/action language, or
- a composite expression that consists of a composer (that belongs to a language) and several subexpressions (where each recursively also belongs to a language – in many cases, the same as the composer).

The language binding is made explicit by the namespace that is used in the root node of an expression (and declared there or in one of its ancestor elements; e.g. in the <eca:rule> element) or by the lang attribute (for opaque fragments). The namespace declaration always yields a URI. The value of the lang attribute may either be a URI, or a namespace prefix that is declared before (and must be resolvd to to URI by the respective application).

The meaning of the URIs will be discussed in Section 3.7 (since there is not yet a standardization what is "behind" the namespace URI, we propose an intermediate solution that is sufficient for the infrastructure in our approach). Each of these languages (i.e., their URIs) has an associated engine that captures the semantics of the (composers of its) language. The engines provide the (expected) interfaces for communication, must keep their own state information, including at least the current variable bindings. Specific tasks of the engines then include e.g. the evaluation of composite events (for the event languages), or the execution of transactions (for the action engines). Thus, the framework itself does not have to deal with actual event detection or transaction execution, but only with employing suitable services (provided by the "owners" of these sublanguages) on the Web.

The leaves of the markup trees are then atomic events, literals, or actions, contributed by the underlying domains (and residing in the domain's ontology and namespace).

Special markup elements are provided for using and binding *variables* in the expressions.

3.4 Communication between Rule Components

Although the languages are heterogeneous wrt. the components, there is a common requirement: to support language heterogeneity at the rule component level, there must be a precise convention between all languages how the different components of a rule can exchange information and interact with each other.
In the following, we assume that every processor implements simple fragments of at least one XML and RDF query language (default for XML: XPath) for working locally with small XML or RDF fragments. Such expressions are used in local XSLT-style select="*expr*" attributes.

3.4.1 Logical Variables

As for XChange, here we also propose to use *logical variables* in the same way as in Logic Programming that can be bound to several things: values/literals, references (URIs), XML or RDF fragments, or events. The respresentation of the bound items must be in ASCII, e.g., URIs, serialized XML, or XML-serialized RDF. The binding of a variable to an event (or a sequence of events) e.g. occurs in XChange, or in *cumulative aperiodic events* in the SNOOP language [25]; they can then be used for extracting values from these events. Variables can be bound by the rule (as constants upon registration) or by the components and used in later components.

- Similar to deductive rules, variables used for communication occur *free* in the components, their scope is the rule,
- While in deductive rules, variables must be bound by a positive literal in the body[1] to

[1] note that there are different terminologies in the literature about "positive" and "negative" literals: In Logic Programming, these notions are defined wrt. the rule body, whereas in works based on resolution of disjunctive clauses, they are defined wrt. those literals (as in Xcerpt/XChange). Since a (Horn) clause $p(x) \vee \neg q(x)$ corresponds to a LP rule $p(x) \leftarrow q(x)$, in the first case, $q(X)$ is a negative binding whereas in the second case,

serve as join variables in the body (adhering to safety requirements!) and to be used in the head, in ECA rules we have four components that induce an information flow according to Figures 3.6 and 3.7; see Section 3.5 for details of the execution semantics.

- Positive occurrences are defined analogously to deductive rules based on the term/formula structure (must be done with the semantics of each individual such language).
- Positive occurrences can be used to bind variables to a value. In case that a variable occurs positively several times, it acts then as a join variable, i.e., the values must coincide; this e.g. allows for an event component that in some cases binds a variable which is then used as a join variable in the condition, and in other cases is only bound by the latter.
- Negative occurrences of a variable *use* the value the variable has been bound *before*.
- Thus, during execution of a rule, any variable occurring negatively must be bound to a value earlier on the rule level (e.g., with the rule's initialization, or by deriving its value from another variable) or in an "earlier" (E<Q<T<A) or at least "earlier" in the same component as where the negative occurrence is. This leads to the usual definition of safety of rules.
- Expressions can also use local variables, e.g., in first-order logic conditions. In this case, the scope of a variable is local, e.g., by a quantifier.
- Variables in the action component: Using variables as parameters to an action in the action component counts as negative occurrences (in case that a language used in the action component that does not define positive or negative occurrences. Thus, for languages used in the action component hat do not define the notions of positive/negative occurrences, all occurrences are negative. Note that this allows for binding a variable in the action component, e.g., by allowing for evaluation of queries in that component, like in Transaction Logic [14] that defines its own notions of positive occurrences.

The relationships between rules, rule components and variables are shown in Figure 3.12):

- for each rule R, R.scopes (the set of all logical variables whose scope is the rule) is the union of R.occurs_positive (before or between evaluating components) and C.free for all its components C;
- for each component C of a rule R, and also for each expression inside any component, C.positive, C.negative, C.free, C.bound, denote the sets of variables that occur positively, negatively, free or bound.

  Free variables occur either positively or negatively:
  $C.\mathsf{free} = C.\mathsf{positive} \cup C.\mathsf{negative}$.

  The information which variables are bound in/by an expression is only relevant for expressions, not for the rule components.

Declaration of Variables. Usually, languages based on logical variables do not use explicit variable declarations. Nevertheless, for illustration, we sometimes specify means for declaring variables a priori. Then, the binding mechanism can also be extended with a type system. The markup may *optionally* contain explicit declarations of variables (see Figure 3.12):

- declaration of variables whose scope is the rule in the eca:rule element,
- declaration of variables to be bound in each of the components, even in subexpressions,

it is a positive binding.

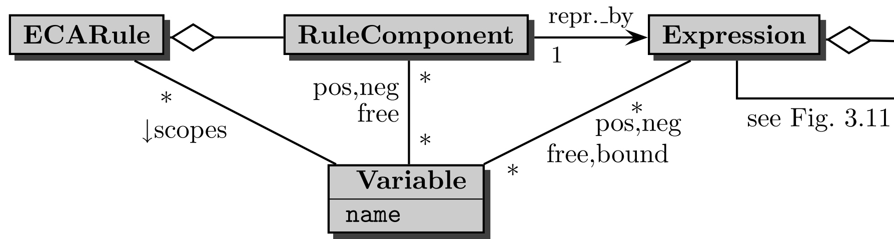


Figure 3.12: Rules and Variables

- declaration of variables to be used in each of the components, even in subexpressions.

However, the goal is that this information can be derived from the markup and the semantic information about the component languages. For this, every service that "offers" a language should provide the following functionality (in addition to DTD etc.):

- Given an XML or RDF fragment of an instance of the language: Validation of the fragment, list of all variables that are used, and all variables that occur positively (i.e., can be bound by this fragment).

3.4.2 Binding and Using Variables

While the semantics of the ECA rules provides the infrastructure for these variables, the markup of specific languages must provide the actual handling of variables (mainly: binding variables) in its expressions. We propose to use a uniform handling of variables in the ECA language, and in E, Q, T, and A component languages.

Variables: Syntax. The ECA-ML language provides constructs for binding variables, where we borrow from XSLT: use variables by {$*var-name*}, and bind them by <variable name="..."> elements:

- <eca:variable name="*name*">
  content
  </eca:variable>

  where *content* is any expression (e.g., an event specification or a query) that returns some value. The variable is then bound to this value (see also Examples 3.3 and 3.6 below).
- <eca:variable name="*name*" lang="xpath" select="*expr*"/>
  can be used for binding a new variable based on already bound ones in *expr*. These expressions can be in any language (e.g. XPath) that an ECA service implements for such simple local evaluations. The above is a shorthand for

  <eca:variable name="*name*">
  <eca:query>
  <eca:opaque lang="xpath"> *expr* </eca:opaque>
  </eca:query>
  </eca:variable>

These variable elements are allowed in ECA-ML anywhere in the upper level of an ECA rule (cf. below description of handling functional results). We propose to use similar constructs also in the component languages, but the actual decision is up to the language designers.

Since we have yet neither presented concrete markup languages for events, conditions and actions, nor concrete means for binding variables to values extracted from atomic events (both will be presented in Section 3.6), we illustrate the use of variables by simple rules with opaque components where each rule component has only a textual, *opaque* specification that is to be understood by some language engine.

Rules with Opaque Components; Example for Variables. Opaque rule components are again marked up as text content of an eca:opaque element that references the language via its lang attribute (see Example 3.3). In this example, information about communicated variables and bindings is given explicitly (and used for renaming variables, when e.g. a language accepts only capitalized variables). Thus, the component elements list all variables that are used or bound by them (i.e., whose bindings must be exchanged with the engine), also optionally giving their names in a use attribute (e.g., for embedding JDBC where variables are only named ?1, ?2 etc.).

Example 3.3 *Consider an ECA rule with opaque components (using different languages) that, whenever a flight is cancelled, sends a message to the destination airport that the flight will not take place:*

```
<eca:rule xmlns:eca="http://www.eca.org/eca-ml"
    xmlns:xpath="http://www.w3.org/XPath"
    xmlns:pseudocode="http://www.pseudocode-actions.nop">
  <eca:variable name="Schedule">http://localhost/schedule.xml</eca:variable>
  <eca:variable name="Flight"/>
  <eca:variable name="Destination"/>
  <eca:variable name="event">
    <eca:event>
      <eca:atomic-event>
        <travel:cancel-flight/>
        <!-- matches any travel:cancel-flight event, e.g.
          <travel:cancel-flight code="LH1234"/> -->
      </eca:atomic-event>
    </eca:event>
  </eca:variable>
      <!-- the matched event is now bound to the variable $event -->
  </eca:variable name="Flight"
        lang="xpath" select="$event/cancel-flight/string(@code)"/>
  flq eca:variable name="Destination">
    <eca:query>
      <eca:opaque lang='xpath'>
        <eca:use-variable name="Flight" use="$Flight"/>
        <eca:use-variable name="Schedule" use="$Schedule"/>
        string(document($Schedule)//flight[@id=$Flight]/@to)
      </eca:opaque>
    </eca:query>
  </eca:variable>
  <!-- evaluates XPath expression and binds the result to the variable 'Destination' -->
```

```
  <eca:test>
    <eca:opaque lang='boolean'>true</eca:opaque>
  </eca:test>
  <eca:action>
    <eca:opaque lang='pseudocode'>
      <eca:use-variable name="Flight"/>
      <eca:use-variable name="Destination"/>
      send "Flight $Flight has been cancelled today" to the destination airport ...
    </eca:opaque>
  </eca:action>
</eca:rule>
```

The ECA engine proceeds as follows: It binds the variable Schedule as a constant to the given value and allocates variables Flight and Destination. The event component consists only of an atomic event. If such an event, e.g. <travel:cancel-flight code="LH1234"/> is detected (by matching), it is bound to the variable event (this semantics coincides with most event detection semantics that return the relevant event sequence as the result). In the next step, the variable Flight is bound by evaluating an XPath expression against the value of event, yielding the binding Flight/"LH1234".

Next, the ECA engine submits the query where ($Schedule is replaced with the constant URL, and $Flight replaced with the actual binding "LH1234") to the XPath engine, that evaluates the expression and returns its result, i.e., the identifier of the destination airport (e.g., "FRA"). The ECA engine binds the returned result to its variable Destination. The condition is then empty (every flight has a destination). Next, the pseudocode fragment in the action component is equipped with the flight number and the destination airport and a message is sent.

3.4.3 Communication of Results and Variable Bindings

For the *framework*, the only important thing is here to define interfaces and a format how to exchange results and variable bindings, i.e., returning results and bound variables from an expression, and to communicate already bound variables down to expressions for using them (as join variables or in actions).

Upward Communication: (Functional) Results and (Logical) Variables. There are several possibilities what the "result" of evaluating a rule component can be:

- Logic-Programming-style languages that bind variables by matching free variables (e.g. query languages like Datalog, F-Logic, XPathLog). Here, the matches can be *literals* (Datalog) or literals and structures (e.g., in F-Logic, XPathLog, Xcerpt). Similar techniques can also be applied to design languages for the event component.
- Functional-style languages that are designed as functions over a database or an event stream:
  - query languages that return a set of data items (e.g., SQL, OQL) that can be interpreted as producing a set of variable bindings (attribute names as variables; probably obtained by the "renaming" operator of the relational algebra),
  - query languages that return a data fragment (e.g. XQuery, Xcerpt – this is only possible since "schema-free" data like XML exists). Here, the result is not bound to an obvious variable name. Note that this should result in a set/sequence of variable bindings if a set/sequence of nodes is created.

– for event languages, the “result” of an expression can be considered the sequence of detected events that “matched” the event expression in an event stream (e.g., XChange).

In this case, the languages can also *use* variables that are bound before. Thus, “downward” communication is explicit, whereas the upward communication is implicit (and the result must be bound to a variable by the *surrounding* language). Note that the answer can even be empty which *sometimes* is interpreted as “false”.

- Both forms can be mixed (F-Logic, cumulative operators in event languages).

To cover all of them, we propose a structure as e.g. used in the Florid system [33] where with *each* result, a set of variable bindings is associated. We propose the following representation for interchange of results and variable bindings:

```
<! ELEMENT answers (answer*)>
<! ATTLIST answers component {event|query} #IMPLIED
            reference CDATA/URI #IMPLIED>
<! ELEMENT answer (result?,variable-bindings?)>
<! ATTLIST answer component {event|query} #IMPLIED
            reference CDATA/URI #IMPLIED>
<! ELEMENT result ANY >
<! ELEMENT variable-bindings (tuple+) >
<! ELEMENT tuple (variable+) >
<! ELEMENT variable ANY >
<! ATTLIST variable name CDATA #REQUIRED
            ref URI #IMPLIED>    <!-- variable has either ref or content-->
```

```
<eca:answers [component="{event|query}"] [ref="string"]>
  <eca:answer [component="{event|query}"] [ref="string"]>
    <eca:result>
      any result structure
    </eca:result>
    <variable-bindings>
      <tuple>
        <variable name="name" ref="URI"/>
        <variable name="name">
          any value
        </variable>
          :
      </tuple>
    </variable-bindings>
  </eca:answer>
  <eca:answer>
    :
  </eca:answer>
</eca:answer>
```

A set of answers consists of multiple answers, where each answer consists of a result value and/or a set of tuples of variable bindings. A variable binding can either be given inline as serialized XML, or as a URI reference (e.g., to a Web page, or an RDF URI). The answers can optionally contain a reference to an ID that in case of asynchrounous communication indicate to

what they are an answer (this is needed for event detection; query handling can be synchronous or asynchronous).
Note the following:

- In cases where only one single answer is produced (which is often the case for event detection, or when calling a "functional" Web Service), the outer <answers> may be omitted, returning only one <answer> structure.
- for services that return no functional result (e.g., a Datalog query service) or no variable bindings (e.g., an XQuery service), each <answer> structure contains only the relevant subelement.
- for services that return only a single functional result (an event sequence, or an answer to an XQuery or SQLX query), it is allowed not to mark it up at all. It is then treated as <result> element of a single answer and can be bound to an ECA-level variable as described below.

For making the functional result part accessible in the ECA rule, it must immediately be bound to a variable (since event detection often returns the relevant event sequence in this way, we illustrate this situation in Section 3.5.1) as described above:

```
<eca:variable name="result-var">
  content
</eca:variable>
```

where *content* is an event specification or a query according to the above discussion of functional semantics of events and queries:

- If the result of *content* is one or more <answer>, then for each <answer>, every <tuple> in the <variable-bindings> part is extended with the variable *result-var* which is bound to the <result> part of the <answer> (see Examples 3.3 and 3.6 and below).
- If the result of *content* is other XML (or analogously for RDF), *result-var* is simply bound to it in the same way as for XSLT (this often saves writing wrappers to the above exchange format).

Note that most currently existing tools (e.g. query interfaces) do not return their data in such a format. In such cases, wrappers (that can be provided locally by the ECA service) can be used. Note that the XQuery return clause (and similar constructs like XML generating functions in SQLX) can be used to return this format directly (this functionality will especially be used in the prototype).

Downward Communication: Variable Bindings. Downward communication of values is needed in the following situations during rule evaluation:

- Variables of parameterized rules that are bound at registration time,
- variables bound in previous components are communicated to queries and actions.

The generic format is given by the above <variable-bindings> element that communicates a set of tuples of variables. As a specific format we propose to support XQuery (e.g. for opaque query components) by a built-in wrapper: by adding let *var* := *xml-fragment* statements in front of the query.

3.5 Semantics of Rule Execution

This section deals with the overall semantics of ECA rules and the abstract semantics for each of the components.

Consider again the above example: For *each* "cancelled flight" event, the rule is "fired". "Fired" means that the event component produces *one* "answer". The next "cancelled flight" message fires another rule instance that will be completely independent.

For that "answer" to the event component, the (one and only) destination is selected, bound to the Destination variable, and for this pair (Flight, Destination), the action is triggered.

A different (and more complex) situation occurs, if the query component produces several answers, e.g. for the following task:

> *Consider the same situation as above, but now notify* every customer *who has a reservation for this flight (e.g., by SMS).*

Comparison: Firing Deductive Rules. For deductive rules (that do not have an event component) in *bottom-up* evaluation, the body is evaluated and produces a set of tuples of variable bindings (Datalog, F-Logic, Transaction Logic, Statelog, XML-QL, XPathLog, Xcerpt; in some sense also the basic form of XQuery). Then, the rule head is "executed" by *iterating* over all bindings, for *each* binding instantiating the structure described in the head (in some languages also executing actions in the head).

The semantics of ECA rules should be as close as possible to this semantics, adapted to the temporal aspect of an event:

ON event AND additional knowledge, IF condition then DO something.

3.5.1 Firing ECA Rules: the Event Component

An event is something that occurs (or, that is detected – in contrast to local databases that represent a closed world, the occurrence of an event somewhere in the Web does not necessarily mean that it is actually detected anywhere where it is relevant). Formally, detection of an event results in an occurrence indication, together with information that has been collected (consider here the data exchange format discussed in Section 3.4.3). The ECA engine must then execute the rule accordingly, using the obtained variable bindings. Note that the cardinality of answers must be considered in this case:

Example 3.4 (Exam Registration) *Consider the following scenario: for an exam, first the (online) registration is opened, then students register, and at a given timepoint, the registration closes. Assume the events to be marked up as e.g.*

<reg_open subject="Databases"/>
<register subject="Databases" name="John Doe"/>
<reg_close subject="Databases"/>

A can e.g. describe an action to be taken "if registration for an exam E is opened, students $X_1, \ldots, X_n$ register, and registration for E closes". The composite event is then formulated as "registration to an exam is closed after (it had been opened and) students $s_1, \ldots, s_n$ registered" and is reported at the timepoint when the registration closes. With its occurrence indication, information about the subject E and registered students $X_1, \ldots, X_n$ is given. Such a cumulative *semantics is provided by appropriate event operators, e.g. by SNOOP [25].*

The rule must then be fired for this one event.

Example 3.5 *Consider the following rule: "If flight F is delayed for more than 25 minutes, do ...". Information about delayed flights is available all 10 minutes as a message including a report of the form*

```
<msg:receive-message sender="service@fraport.com">
  <msg:content>
    <travel:delayed-flight flight="LH1234" time="30"/>
    <travel:delayed-flight flight="AF0815" time="90"/>
    <travel:delayed-flight flight="CY42" time="60"/>
      :
    <travel:cancelled-flight flight="AL4711"/>
    :
  </msg:content>
</msg:receive-message>
```

There are two ways how to maintain this rule:

- *XML level: the rule designer knows that such messages come in, and formulates a suitable atomic event pattern (cf. Section 3.6.1 as*

  ```
  <eca:rule>
    <eca:event>
      <eca:atomic-event>
        <receive-message sender="service@fraport.com">
          <content>
            <delayed flight="$flight" time="$minutes"/>
          </content>
      </eca:atomic-event>
    </eca:event>
      :
  </eca:rule>
  ```

  which binds variables flight *and* minutes*.*

- *Formulating the rule purely in the application domain as*

  ```
  <eca:rule>
    <eca:event>
      <eca:atomic-event>
        <delayed flight="$flight" time="$minutes"/>
      </eca:atomic-event>
    </eca:event>
  </eca:rule>
  ```

  and using an ECE (see Section 3.1.5) event derivation rule "if there is a message whose content matches event E, then consider E to be detected".

The answer from the event detection module can result in

- *one answer, containing several tuples, or*
- *several answers, containing one tuple.*

Note that later, the same event pattern will be detected again, which again fires the rule.

In case that an occurrence indication contains *multiple* tuples of variable bindings, the semantics must be carefully considered: The tuples must be regarded as semantically independent since they –although "just by chance" detected at the same time– represent independent events. For that reason, a correct (but not always most efficient) semantics would be that the ECA engine immediately separates them and fires independent instances of the rule (see Sections 3.5.4 and 3.8).

Thus, if a final event should report about something set-like, this must be contained in the semantics of the event language, not of the ECA language – then it is practically too late. As stated above, an explicitely cumulative semantics is e.g. supported by the SNOOP event algebra.

In many approaches, the "result" of event detection is the sequence of the events that "materialized" the event pattern to be detected. In this case, an appropriate way is to bind this result to a variable as shown above and afterwards the values of other variables can be extracted from this one.

Example 3.6 *Consider the following situation from Example 3.4 and an event specification (using XPathLog and regular expression syntax in an obvious way):*

```
<eca:rule ... >
  <eca:variable name="Subj"/>
  <eca:variable name="regseq">
    <eca:event>
      <eca:opaque lang='xpathlog-events'>
        <eca:bind-variable name="Subj"/>
        reg_open[@subject→Subj], register[@subject→Subj]*, reg_close[@subject→Subj]
      </eca:opaque>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

The returned information from the event detection service in the markup proposed in Section 3.4.3 looks as follows, returning the relevant event sequence. The variable Subj *has also been bound in the event component:*

```
<eca:answer component="event" ref="identifier">
  <eca:result>
    <reg_open subject="Databases"/>
    <register subject="Databases" name="John Doe"/>
    <register subject="Databases" name="Scott Tiger"/>
      :
    <reg_close subject="Databases"/>
  </eca:result>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="Subj">Databases</eca:variable>
```

```
    </eca:tuple>
  </eca:variable-bindings>
</eca:answer>
```

Next, the variable regseq *is bound to the* <result> *part. The variable bindings after completely evaluating the event component look as follows:*

```
<eca:variable-bindings>
  <eca:tuple>
    <eca:variable name="regseq">
      <reg_open subject="Databases"/>
      <register subject="Databases" name="John Doe"/>
      <register subject="Databases" name="Scott Tiger"/>
        :
      <reg_close subject="Databases"/>
    </eca:variable>
    <eca:variable name="Subj">Databases</eca:variable>
  </eca:tuple>
</eca:variable-bindings>
```

3.5.2 The Query Component

This second component is concerned with *static* information that is obtained and restructured from two areas:

- analyzing the data that has been collected by the event component (in the variable bindings), and
- based on this data, stating queries against databases and the Web.

Whereas the event component of a rule may be "answered" by detecting occurrences of the event pattern several times, the query component returns all answers at the same time. The query component is very similar to the evaluation of database queries and rule bodies in Logic Programming: in general, it results in a set of tuples of variable bindings (that are possible answers to a query).

Grouping: Set-Valued vs. Multi-Valued. An important issue here is to deal with sets (e.g., in the above examples, all customers who booked a flight that has been cancelled, or all students that registered for an exam):

- bind a variable to a collection, e.g.,
  $\beta = \{\text{Subj} \rightarrow \text{'Databases'},\ \text{student} \rightarrow \{\text{'John Doe'}, \text{'Scott Tiger'}, \ldots\}\}$, or
- produce separate tuples of variable bindings:
  $\beta_1 = \{\text{Subj} \rightarrow \text{'Databases'},\ \text{student} \rightarrow \text{'John Doe'}\}$,
  $\beta_2 = \{\text{Subj} \rightarrow \text{'Databases'},\ \text{student} \rightarrow \text{'Scott Tiger'}\}$.

We follow again the Logic Programming specification, followed and implemented also in Prova (cf. Appendix B), that every answer produces a variable binding. For variable binding by matching (as in Datalog, F-Logic, XPathLog, Xcerpt etc.), this is obvious. Since we also allow variable bindings in the functional XSLT style, the semantics is adapted accordingly:

- each answer node of an XPath expression yields a variable binding;
- each node that is *return*ed by an XQuery query yields a variable binding; if the XQuery query is of the form
  <*name*>{ for ... where ... return ...} </*name*> ,
  then the whole result yields a single variable binding.

Example 3.7 *Consider again Example 3.6 where the resulting event contained several registrations of students. For doing anything useful, their names have to be extracted.*

1. *as multiple string-valued variables:*

```
<eca:rule ... >
    :
  same as above, binding variables "Subj" and "regseq"
    :
  <eca:variable name="Student">
    <eca:query>
      <eca:opaque lang='xpath'>
        $regseq//register[@subject=$Subj]/@name/string()
      </eca:opaque>
    </eca:query>
  </eca:variable>
    :
</eca:rule>
```

   The above query generates the extended variable bindings
   $\beta_1 = \{\mathit{Subj} \rightarrow \mathit{'Databases'}, \mathit{regseq} \rightarrow \mathit{(as\ above)},\ \mathit{Student} \rightarrow \mathit{'John\ Doe'}\}$,
   $\beta_2 = \{\mathit{Subj} \rightarrow \mathit{'Databases'},\ \mathit{regseq} \rightarrow \mathit{(as\ above)},\ \mathit{Student} \rightarrow \mathit{'Scott\ Tiger'}\}$.

2. *or as a single variable:*

```
<eca:rule ... >
    :
  same as above, binding variables "Subj" and "regseq"
    :
  <eca:variable name="Students">
    <eca:query>
      <eca:opaque lang='xquery'>
        <students>
          for $s in $regseq//register[@subject=$Subj]/@name/string()
          return <name> { $s } </name>
        </students>
      </eca:opaque>
    </eca:query>
  </eca:variable>
    :
</eca:rule>
```

   This query generates the extended variable binding

$$\beta = \{\ \textit{Subj} \rightarrow \textit{'Databases'},\quad \textit{regseq} \rightarrow \textit{(as above)},\ \textit{Students} \rightarrow \textit{<students> <name>John Doe</name> <name>Scott Tiger</name> </students>} \}$$

The above query showed how data from the variable bindings obtained from the event detection is extracted. Note that this query is very similar to the *event queries* mentioned in XChange (instead of the above opaque query, also an Xcerpt/XChange query could have been used).

The next example shows a query against an XML repository in the Web. Again, the answer of the query component is represented in a single XML variable. This example then leads immediately to questions about handling the action component.

Example 3.8 *Consider an ECA rule with opaque components (using different languages) that, whenever a flight is cancelled, notifies* every customer *who has a reservation for this flight (e.g., by SMS), and sends a message to the airport hotel with the names of* all customers *to make a pre-reservation for this night.*

```
<eca:rule xmlns:eca="http://www.eca.org/eca-ml"
    xmlns:datalog="http://www.lp.org/datalog"
    xmlns:xpath="http://www.w3.org/XPath"
    xmlns:pseudocode="http://www.pseudocode-actions.nop">
  <eca:variable name="Bookings"> http://localhost/schedule.xml </eca:variable>
  <eca:variable name="Flight"/>
  <eca:event>
    <eca:opaque lang='datalog'>
      <eca:bind-variable name="Flight"/>
      flight_cancellation(Flight)   <!-- matches Flight against received message -->
    </eca:opaque>
  </eca:event>
  <eca:variable name="Customers">
    <eca:query>
      <eca:opaque lang="xquery">
        <eca:use-variable name="Flight"/>
        <eca:use-variable name="Bookings"/>
        return
          <customers>
          { for $c in document($Bookings)//flight[@id=$Flight]/reservation/customer
            return $c }
          </customers>
      </eca:opaque>
    </eca:query>
  </eca:variable>
  <!-- evaluates XPath expression and binds each of the results to the variable 'Customers' -->
  <eca:test>
    <eca:opaque lang='xpath'>
      $Customers/customer
    </eca:opaque>
  </eca:test>
  <eca:action>
    <eca:opaque lang='pseudocode'>
      <eca:use-variable name="Customers"/>
      <eca:use-variable name="Flight"/>
```

```
      send one message with all Customers/customer/name to the hotel,
      then
      for each N in Customers/customer/@phonenr do
        notify_cancellation(Flight, sms:N)
    </eca:opaque>
  </eca:action>
</eca:rule>
```

In the first case, the action has a multi-valued semantics, whereas in the second case it has a set-valued semantics. This simple case could be solved by binding the *set* of customers to a variable. The above "solution" moves the solution inside the query component, and is sufficient in this case, but not in general (and less declarative).

3.5.3 The Test Component

The test component is still concerned with the information obtained so far. It evaluates a condition which is a mapping from a knowledge base to true/false. In general, the evaluation of conditions is based on a logic over literals with boolean combinators and quantifiers. A Markup Language exists with FOL-RuleML [11]. Since first-order logic is in general undecidable, it is recommended to use suitable fragments. Instead of first-order atoms, also "atoms" of other data models can be used. Additionaly, we envisage to allow to use simple expressions like XPath of a language that is locally supported in the ECA engine. Note that XPath expressions are also literals that result in a true/false (true if the result set is non-empty) value (as in the above example). Variables are communicated to the test in the same way as above. The test component returns then the set of tuples that satisfy the condition (for further propagation to the action component).

3.5.4 Summary of Event, Query and Test Semantics

Given the variable bindings resulting from the event component, the semantics of evaluating queries is based on the usual answer & join semantics of Logic Programming: for the cases where queries only contain positive occurrences of variables, the resulting variable bindings of the event and query components are just the join of the individual sets of variable bindings. As long as only positive queries are used, the semantics of the query component is commutative. In case of negative occurrences, the usual safety constraints apply, variables must be bound positively before they can be used in a negated occurrence. Then, negation is interpreted as set difference. The test component acts as a selection that removes all variable bindings that do not satisfy the test.

The normal form induces a sequential operational semantics; other evaluation and execution strategies are possible based on equivalence transformations (see Section 3.8).

In each stage, the variable bindings are considered as a set of tuples that is represented in XML as above, and communicated with the event, query and test services. Note that this allows for aggregation and grouping constructs in queries and tests.

Multiple Occurrences. The "plain" semantics of an ECA rule assumes that a rule is fired for each individual occurence of the event pattern given in its event component. As shown above, that multiple independent instances of an event pattern are detected at the same time, and reported as a whole. In such cases, it is possible to fire the rule for all these occurrences together

(since we do not bind program variables, but use the notion of logical variables and sets of tuples of variable bindings, the underlying declarative semantics allows for this). Semantically, there is a only a difference if transactional issues in the action component are considered. In this case it is enough to separate the instances when executing the action component.
Nevertheless, some efficiency aspects have to be considered; see Section 3.8.

3.5.5 The Action Component

The action component is the one where *actually* something is done in the ECA rule: for each variable binding, the action component is executed. The action component may consist of several <eca:action> elements which can use different action languages. The semantics is that all actions are executed. Note that actions are not allowed to bind variables, thus they are independent on this level. (Note that sequential and conjunctive execution of actions can also be specified on the level of action languages *inside* the <eca:action> element.)

Grouping and de-grouping execution steps. Here, the possibility of grouping and de-grouping is required: it makes a strong difference if a set is represented by one tuple of variables, or the whole set is bound as a set in one tuple.

Example 3.9 *Consider again Example 3.6 where the names of students that have registered for an exam have been collected:*

1. *In the first case, we had the following variable bindings:*
   $\beta_1 = \{\mathit{Subj} \to \text{'Databases'}, \mathit{regseq} \to \text{(as above)}, \mathit{Student} \to \text{'John Doe'}\}$,
   $\beta_2 = \{\mathit{Subj} \to \text{'Databases'}, \mathit{regseq} \to \text{(as above)}, \mathit{Student} \to \text{'Scott Tiger'}\}$,
   i.e., a set of two tuples.

   If the action now is "(for each tuple), send the lecturer a mail with the value of the variable Student", the lecturer will get two mails, each one with one student.

   Instead, we want to send the lecturer one *mail with the names of all registered students.*
2. *For such cases, a functionality for "group by Subject" (which results in only one tuple) would be useful, e.g. a way to group by Subj, collecting all names in a list (and forgetting about regseq, resulting in*
   $\beta_{grp} = \{\mathit{Subj} \to \text{'Databases'}, \mathit{Student} \to \{\text{'John Doe'}, \text{'Scott Tiger'}\}\}$,
   for which the action then can be called.
3. *In the second case, we have only one variable binding (very similar to β_{grp} above,*

   β = { { *Subj* → *'Databases'*, *regseq* → *(as above)*,
   Students → <*students*> <*name*>*John Doe*</*name*>
   <*name*>*Scott Tiger*</*name*> </*students*>}

   for which the action can be called immediately (and e.g. submit the XML structure of Students as the message content).

On the other hand, if the task is "to send a message to each of the registered students", the first case is immediately suitable, whereas the second one needs either an iteration inside the action component ("for each name in Students do ...") or, – more declaratively on the ECA level – an ungrouping*, resulting in the variable bindings of β_1 and β_2.*

Such cases are very frequent. A rule can even contain actions of both kinds, e.g., in

For the above E&Q components, “send the lecturer an e-mail with the names of all students *and* send a confirmation message to each of the registered students”.

In such cases, the number of variable bindings must be changed by grouping or de-grouping, dependent on what the result of the E&Q components are. Grouping and ungrouping on the ECA level is allowed *before* each <eca:action> element. We propose to inherit these elements also to the action languages:

- all tuples that coincide in *all* named variables are grouped together; the other variables are aggregated as lists, sum, avg etc., or omitted (a default can be specified; *aggr-op* = list|sum|avg|omit|...):

```
<eca:group-by aggr="aggr-op">
  <eca:group-variable name="name"/>
    :
  <eca:group-variable name="name"/>
  <eca:aggr-variable op="aggr-op" name="name" aggr-name="name"/>
    :
  <eca:aggr-variable op="aggr-op" name="name" aggr-name="name"/>
</eca:group-by>
```

- Flattening a list or sequence: The new variable binding is obtained by splitting a list, or applying a query to the variable (XPath, or any default language configured by the ECA service or the rule). Note that this can also be done by a sequence of <eca:query> elements.

```
<eca:ungroup-by>
  <eca:ungroup-variable>variable-name
    [<eca:ungroup-query> ...</eca:ungroup-query>]
  </eca:ungroup-variable>
    :
  <eca:ungroup-variable>variable-name
    [<eca:ungroup-query> ...</eca:ungroup-query>]
  </eca:ungroup-variable>
</eca:ungroup-by>
```

3.5.6 Transactions

Although the issue of transactions does not directly have to do with the semantics of ECA rules, some issues should be raised here. Transactional issues are only concerned with the action component (events can neither fail nor be rolled back, queries and tests can also not “fail” and there is nothing to be rolled back). Transactional functionality can be offered independently by the action languages *inside* the <eca:action> elements.

Since we stated above that the semantics for excution of the action component is the same as for executing the head of a deductive rule, i.e., handling it separately for each tuple of variable bindings, transactions that should cover the whole group must explicitly be expressed on the ECA level.

For this, we propose an

```
<eca:transaction attributes> ... </eca:transaction>
```

element that can occur around or anywhere in an <eca:action> element.

With this, e.g., actions for *all* tuples can be grouped as a transaction by

```
<eca:rule ...>
  <eca:event> ...</eca:event>
  <eca:query> ...</eca:query>
  <eca:test> ... </eca:test>
  <eca:transaction>
    <eca:action> ... </eca:action>
  </eca:transaction>
</eca:rule>
```

A further element allows to take a group of tuples together for execution of a transaction (note that in contrast to <eca:group-by>, the number of tuples does not change):

```
<eca:transaction-group-by/>
  <eca:group-variable name="name"/>
    :
  <eca:group-variable name="name"/>
</eca:transaction-group-by>
```

(e.g., if an event contains a list of delayed flights, the query component returns a pair of variable bindings for each customer, and the appropriate actions should be separate transactions for each flight).

3.6 Languages for the Event, Query, Test, and Action Components

In this section, we illustrate each of the rule components by well-known sample languages. Since query languages and languages for tests are well-known (e.g., XQuery, first-order logic), we sketch these aspects only shortly, and focus on the event component and the action component, taking as sample languages the SNOOP event algebra [25], and the CCS process algebra [52]. Simple instances can also be obtained by straightforward markup versions for languages like ECA-for-XML, RDFTL etc. that have been mentioned in Section 3.1.2.

3.6.1 Event Component

The event component language is embedded in the language hierarchy as shown in Figure 3.9. It is based on the event ontology as discussed in Section 3.1.4 and allows to express the events according to the ontology of atomic events given in Figure 3.2 and provides the specification of composite events by means of event algebra as shown in Figure 3.11.

Thus, each language used in the *event component* is made up as a combination of one ore more event algebras, using atomic events of one or more applications, and possibly atomic data-level events from several data models, and atomic events from application-independent services. The structure of the event component is shown in Figure 3.13.

Since the usual semantics of evaluating an event algebra expression is to return the matching event sequence, subexpressions from different algebras can be combined easily.

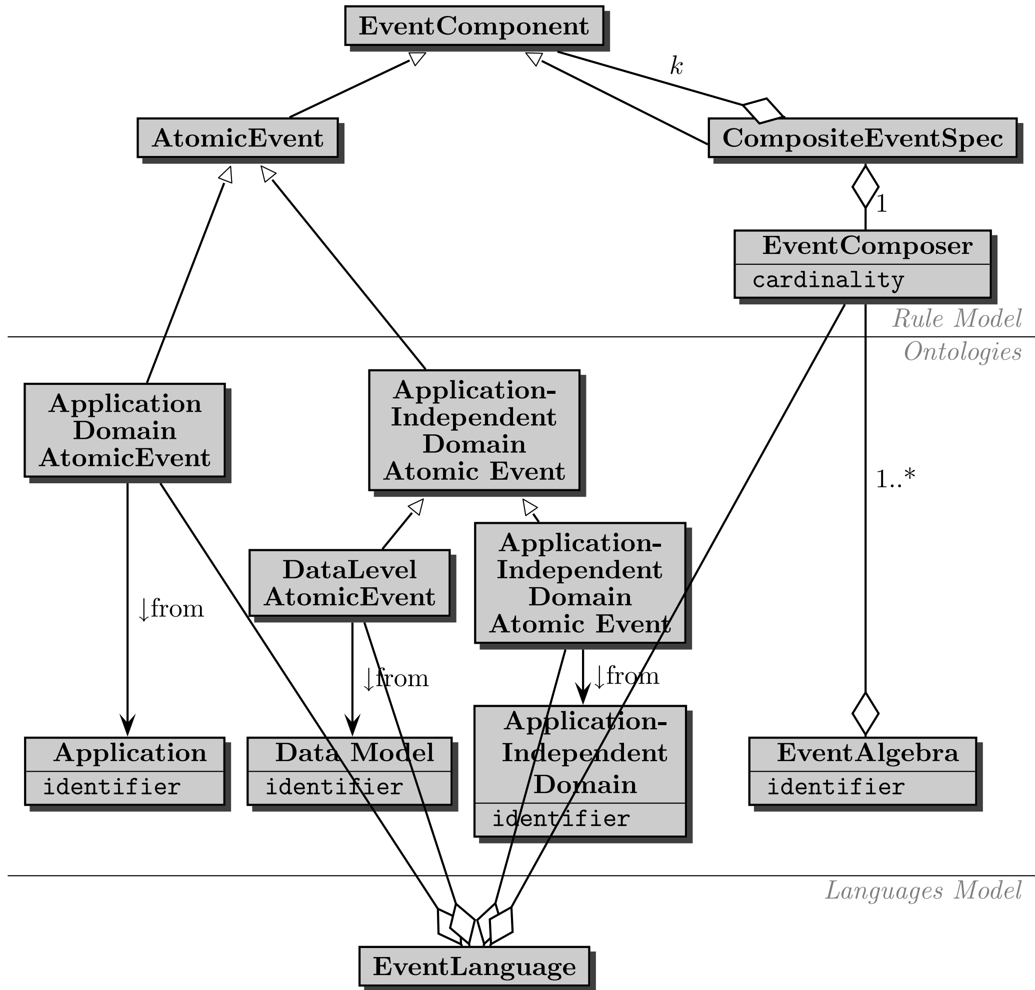


Figure 3.13: Event Component Ontology

3.6.1.1 Handling Atomic Events

The atomic events are the simplest case of event components, and they are also the leaves of event algebra expressions. They do not belong to the namespace of any event algebra, but reside in the domain namespaces. We assume that events are available in XML or RDF format.

Example 3.10 (Atomic Events) *Events are data fragments that are available in XML markup or as RDF fragments.*

- *The events*
  <travel:cancel-flight code="LH1234"/>
  <travel:cancel-flight><travel:code>LH1234</travel:code></travel:cancel-flight>
  are (using different markups) events in the travelling domain that mean that the flight

"LH1234" is cancelled. Note that more information (e.g., that this concerns today's flight that should depart in one hour) must then be accessible from the context. Complete information would be available if the event is of the following form <travel:cancel-flight code="LH1234" date="10062005"/>.

- *The following events have already been used above in an example:*
  <uni:reg_open subject="Databases"/>
  <uni:register subject="Databases" name="John Doe"/>
- *in RDF, events are resources of a type "event" that also have a name and are connected to other resources as parameters.*

In the event component of a rule, either inside of a composite event markup, or when the rule reacts on an atomic event, we propose to use

```
<eca:atomic-event [xmlns:domain="namespace"]>
  contents
</eca:atomic-event>
```

elements to denote that "leaves" of the event language level (instead of eca, also the namespace of the event algebra can be used) are reached; optionally, the namespace of the domain of the included atomic event can be defined here. Inside of <eca:atomic-event> elements, the namespaces of the applications are used with the data *about* the events to be matched.

In general, the event component states some requirements on the events on which a reaction should be taken, e.g., the name of the event, or also its contents (i.e., its parameters). Since the event is seen as an XML or RDF fragment, these conditions can be stated as queries against this fragment.
We investigate two principal ways then to detect and query atomic events of interest (see examples below):

- pattern-based (e.g. by XML patterns, or like in Xcerpt or XML-QL). In this case, inside this element, the actual atomic event is specified and variables can be bound to fragments. XSLT style eca:variable elements or variable references of the form $*var-name* can be used inside the pattern to express that a fragment of the event is bound or must match a variable:

  ```
  <eca:atomic-event>
    <travel:cancel-flight code="$flight"/>
  </eca:atomic-event>
  ```

  or, in the other markup,

  ```
  <eca:atomic-event>
    <travel:cancel-flight>
      <eca:variable name="flight">
        <travel:code/>
      </eca:variable>
    </travel:cancel-flight>
  </eca:atomic-event>
  ```

  matches any travel:cancel-flight event and binds the variable flight to the code of the flight (in the first case, only the string value, in the second case the whole element). In case that flight is already bound, this acts as a (join) condition on the code of the cancelled flight.
- navigation-based: inside the <eca:atomic-event> element, the event itself (as an XML fragment) is available as $event. Then,

```
<eca:test lang="xpath" condition= "xpath-expr"/>
```

can be used for tests (note that this is the same eca:test as for the test component; instead of XPath, other simple languages are possible), and elements of the form

```
<eca:variable name="var-name" lang="xpath"
              select="$event/relative-expr..."/>
```

can be used to access data within the event. Variables can also be addressed by {$*var-name*} as in XQuery (for using them as join variable or for binding them to the matched value). Here, the first variant looks as follows:

```
<eca:atomic-event>
  <eca:test lang="xpath" condition="$event/*/name()='cancel-flight'"/>
  <eca:variable name="flight" lang="xpath" select="$event/cancel-flight/@code"/>
</eca:atomic-event>
```

or

```
<eca:atomic-event>
  <eca:test lang="xpath" condition="$event/*/name()='cancel-flight'"/>
  <eca:test lang="xpath" condition="$event/cancel-flight/@code=$flight"/>
</eca:atomic-event>
```

- both ways can also be mixed:

```
<eca:atomic-event>
  <travel:cancel-flight/>
  <eca:variable name="flight" lang="xpath"
        select="$event/cancel-flight/@code">
</eca:atomic-event>
```

Working with the notion of context nodes should also be considered. Then, the current context node should always be the current position in the event structure (similar to tree-walking in XSLT):

```
<eca:atomic-event>
  <eca:test lang="xpath" condition="./*/name()='cancel-flight'"/>
  <eca:test lang="xpath" condition="./*/@code=$flight"/>
</eca:atomic-event>
```

or

```
<eca:atomic-event>
  <travel:cancel-flight>
      <travel:code>
        <eca:variable name="flight" lang="xpath" select="text()"/>
      </travel:code>
  </travel:cancel-flight>
</eca:atomic-event>
```

Which of the above ways will be supported (or all of them) will be decided in the future.

3.6.1.2 Composite Events: Event Algebras

Event algebras, well-known from the Active Database area, serve for specifying *composite* events by defining *terms* formed by nested application of composers over *atomic* events. There are sev-

eral proposals for event algebras, defining different composers. Each composer has a semantics that specifies what the composite event means.

For dealing with composite events in the context of the ECA rules proposed here, we propose at least the following composers: "E_1 `OR` E_2", "E_1 `AND` E_2" (in arbitrary order), and "E_1 `AND THEN` E_2 `[AFTER PERIOD` $\{< \mid >\}$ *time*`]`" the latter one composing two events and using an additional parameter *time*, indicating the time that has passed between the occurrence of E_1 and E_2. Detection of a composite event means that its "final" atomic subevent is detected:

$$\begin{array}{lll} (1) & (E_1 \nabla E_2)(t) & :\Leftrightarrow\ E_1(t) \vee E_2(t)\ , \\ (2) & (E_1 \triangle E_2)(t) & :\Leftrightarrow\ \exists t_1 \leq t \leq t_1 : (E_1(t_1) \wedge E_2(t)) \vee (E_2(t_1) \wedge E_1(t)). \\ (3) & (E_1 ;_{\Delta t} E_2)(t) & :\Leftrightarrow\ \exists t_1 \leq t \leq t_1{+}\Delta t : E_1(t_1) \wedge E_2(t). \end{array}$$

Event algebras contain not only the aforementioned straightforward basic connectives, but also additional operators. A bunch of event algebras have been defined that provide also e.g. "negated events" in the style that "when E_1 happened, and then E_3 but not E_2 in between", "periodic" and "cumulative" events, e.g., in the SNOOP event algebra [25] of the "Sentinel" active database system. Richer forms of composite events have also been described for XChange (see Chapter 2). [8] considers a special kind of composite events on XML data that are formed by XML update operation events, dealing e.g. with updates concerning different subtrees of an element.

Example 3.11 (Cumulative Event, [25]) *A "cumulative aperiodic event"*

$$A^*(E_1, E_2, E_3)(t)\ :\Leftrightarrow\ \exists t_1 \leq t : E_1(t_1) \wedge E_3(t)$$

occurs with E_3 and reports the collected occurrences of E_2 in the meantime. Thus, its detection is defined as "if E_1 occurs, then for each occurrence of an instance of E_2, collect it, and when E_3 occurs, report all collected occurrences (in order to do something)".

A cumulative periodic event can be used for "after the end of a month, send an account statement with all entries of this month":

$$\begin{array}{l} E(Acct) := \\ A^*(\mathit{first_of_month}(m), (\mathit{debit}(Acct,Am) \nabla \mathit{deposit}(Acct,Am)), \mathit{first_of_month}(m{+}1)) \end{array}$$

where the event occurs with first_of_next_month. The "result" of the expression is the list of all contributing events (cf. discussion in Section 3.4 and below).

XML Markup for the Event Component. The <eca:event> elements contain elements according to event algebra languages. Every subexpression is associated by its namespace with the appropriate components of the language – i.e., an event algebra or atomic expressions from underlying domains. In general, if an event algebra supports an XML markup, it will define its own ways for dealing with atomic events and variables. We propose the markup elements described below for a *unified* approach.

Example 3.12 *The cumulative event from Example 3.6 (there given as a regular expression) can be given in SNOOP as*

$$\mathcal{A}^*(reg_open(Subj), register(Subj, Stud), reg_close(Subj))\ .$$

The following markup binds the complete sequence to regseq *and the subject to* Subj*:*

```
<eca:rule ... >
  <eca:variable name="Subj"/>
  <eca:variable name="regseq">
    <eca:event xmlns:xmlsnoop="http://xmlsnoop.nop">
      <xmlsnoop:cumulative>
        <xmlsnoop:atomic>
          <uni:reg_open>
          <xmlsnoop:variable name="Subj"
              lang="xpath" select="./@Subject" />
          </uni:reg_open>
        </xmlsnoop:atomic>
        <xmlsnoop:atomic>
          <uni:register subject="$Subj"/>
        </xmlsnoop:atomic>
        <xmlsnoop:atomic>
          <uni:reg_close subject="$Subj"/>
        </xmlsnoop:atomic>
      </xmlsnoop:cumulative>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

Note the similarity to XChange's "*Event queries*" that also work on an XML markup of events, but which is restricted to its own set of event combinators, whereas the above mechanism is even parametric in the event combinators – each event algebra may implement its own event detection using this markup proposal.

Example 3.13 *The following specifies, in an illustrative, non-normative (XML) markup, an event for (very simplified) detection of a late train. It is a composite event in the SNOOP (algebraic) language, and uses atomic events from messaging and the domain of train travels. The detection of late trains is made either by being warned by mail the travel agency, or by the occurrence of a domain-specific event signaling changes in a given (pre-defined) source with expected arrival times:*

When a customer registers the rule, the values for the variables myTrain *and* myTravelAgent *have to be supplied.*

```
<eca:rule xmlns:msg="http://www.messages.msg/messages"
    xmlns:mytravel="http://www.trains.org">
  <eca:event xmlns:xmlsnoop="http://xmlsnoop.nop">
    <xmlsnoop:disjunctive>
      <xmlsnoop:atomic>
        <msg:receive-message sender="$myTravelAgent">
          <msg:content>
            <delayed train="$myTrain">
              <xmlsnoop:variable name="newArrival"
                  lang="xpath" select="./@arrivalTime"/>
              </xmlsnoop:variable>
            </delayed/>
```

```
        </msg:content>
      </msg:receive-message>
    </xmlsnoop:atomic>
    <xmlsnoop:atomic>
      <xmlsnoop:cond lang=“xpath” test=“$event/*/name()=’mytravel:changeTime’“/>
      <xmlsnoop:cond lang=“xpath” test=“$event/*/@trainId=$myTrain“/>
      <xmlsnoop:variable name=“newArrival”
          lang=“xpath” select=“$event/*/@newTime” />
    </xmlsnoop:atomic>
  </xmlsnoop:disjunctive>
</eca:event>
<eca:action> an action specification in any markup </eca:action>
</eca:rule>
```

The ECA rule engine registers the whole event *component at the SNOOP service (identified by the URL of the* xmlsnoop *namespace to which the outer element of the event component refers). This composite event is an "or" of two atomic events: the first one is receiving a message (marked-up in XML) with an attribute* sender *which is equal to the value of the variable* myTravelAgent*, and with a* content *with a* delayed *element with an attribute* train *coinciding with that of* myTrain*, using the* xml-pattern *matching mechanism. If so, the variable* newArrival *is bound to the value of the attribute* arrivalTime *of that* delayed *element. The second one is a domain-specific event* travel:changeTime *(that occurs "somewhere in the Web" and has to be detected by Semantic Web mechanisms). It is implicitly bound to* $event*. The details are then checked by XPath expressions against* $event*: If its attribute* trainId *equals the value of the variable* myTrain*, then* newArrival *is bound to the value of the* newTime *attribute of the event.*

Note that here the event sequence itself (that is returned by the SNOOP event detection service) is not bound to a variable – all relevant information can be extracted without this.

Next, we show how a certain event algebra service can e.g. extend the above concepts. In Example 3.12, we illustrated SNOOP's cumulative event. For collecting all names, the event sequence has been kept in a variable regseq and was analyzed later by a query against regseq as illustrated in Example 3.12. An event algebra can e.g. allow to extract a list immediately:

Example 3.14 *Consider again the event of Example 3.11. Atomic events are (i) temporal events that are assumed to be provided/signalled by some service, e.g. as*

```
<temporal:first-of-month month=“5” year=“2005” />
```

and (ii) events of the banking application, provided as e.g.,

```
<banking:deposit account=“1234” >
  <amount>200</amount>
</banking:deposit>
```

The rule is registered (e.g. by a customer) for a given value of $account*.*

```
<eca:rule xmlns:banking=“http://www.banking.nop”
    xmlns:temporal=“http://www.some.webservice” >
  <eca:event xmlns:xmlsnoop=“http://xmlsnoop.nop” >
    <xmlsnoop:cumulative-event cumulative-collect=“list” >
      <xmlsnoop:cumulative-start>
        <xmlsnoop:atomic>
          <temporal:first-of-month>
```

```
        <eca:variable name="month" lang="xpath" select="$event/*/@month"/>
      </temporal:first-of-month>
    </xmlsnoop:atomic>
  </xmlsnoop:cumulative-start>
  <xmlsnoop:cumulative-collect>
    <xmlsnoop:disjunctive>
      <xmlsnoop:atomic> <banking:debit account="$account"/>
        <xmlsnoop:variable name="list" lang="xpath" select="$event"/>
      </xmlsnoop:atomic>
      <xmlsnoop:atomic> <banking:deposit account="$account"/>
        <xmlsnoop:variable name="list" lang="xpath" select="$event"/>
      </xmlsnoop:atomic>
    </xmlsnoop:disjunctive>
  </xmlsnoop:cumulative-collect>
  <xmlsnoop:cumulative-end>
    <xmlsnoop:atomic>
      <temporal:first-of-month month="$month+1"/>
    </xmlsnoop:atomic>
  </xmlsnoop:cumulative-end>
 </xmlsnoop:cumulative-event>
</eca:event>
  :
</eca:rule>
```

Note that here we assume that SNOOP provides the possibility to define the variable list *to be cumulative, i.e., for each*

<**xmlsnoop:variable** name="list" select="$event"/>,

the event (as debit/deposit XML element) is appended*.*

For this, the internal event detection mechanism is required to assign the variable list *several times in an incremental way (such things are allowed by the framework; it is only required that the evaluation returns variable bindings). In an XML setting, the "list" should just be a sequence of nodes, as e.g. generated by an XQuery* let *statement.*

Input to the Event Detection Engine.

For each event algebra to be used in this framework, an event detection engine is required, e.g., as a Web Service that provides the following functionality (see Section 3.7 for considerations on the architecture):

- register an event description as an algebra expression using atomic events from application domains,
- optionally values of already bound variables (by the rule, or when different event languages are used in a tree),
- atomic events – either they are delivered to the detection engine, or it must get them from "somewhere".

  In classical databases, for each event, the form of the event (an operation on a tuple) and the schema of their *information* was clear (e.g., a student registration number); whereas in the Semantic Web, this can be more flexible. On the other hand, with XML and RDF, there are two flexible data formats at hand that allow for representing this information.

Matching of Atomic Events.

The matching of atomic events with the patterns specified by <eca:atomic-event> elements is not specific to an event algebra. This matching can either be implemented by the algebra engine, by the ECA engine (which also relies on such functionality in case of rules whose event component is an atomic event), by the domain services, or by separate services.

Detection of Composite Events.

The internals of the event detection engine are then concerned with implementing the semantics of the event combinators, which can be done in different ways:

- Operators as Classes – (cf. RuleCore, Appendix A),
- Tree and event queries – (cf. SNOOP [25], XChange, Chapter 2),
- Automata and Petri Nets – ODE and SAMOS
  (here, also a representation of automata states in XML and transformations by XSLT can be used),
- RDF: describe how an event description transforms into another upon an atomic event. This semantic solution would require an ontology of event combinators, but allows then for a very high-level specification of rules.

Existing (sub)languages.

Especially, existing tools can be employed in a service-oriented architecture:

- XChange's event query mechanism (cf. Chapter 2). Then, the event component is e.g. marked up by

```
<eca:rule xmlns:banking="http://www.banking.nop"
    xmlns:temporal="http://www.some.webservice">
  <eca:event xmlns:xchange="http://xcerpt.org/xchange">
    <!-- xchange fragment in opaque or xml-markup form -->
  </eca:event>
    :
</eca:rule>
```

- RuleCore (cf. Appendix A) is an ECA system that provides an event detection component where new operators can be added by via appropriate classes. Then, RuleCore can be used to implement and experiment with arbitrary event algebras.

3.6.2 Queries in ECA Rules for the Web

Queries are stated in query languages, like XQuery (where an XML markup proposal has been presented in [78]), F-Logic, RDFQL, XPathLog, or Xcerpt. Note that queries that bind (join) variables can already be used for restricting the results. Queries that are only concerned with the contents a certain database node can use the query language of this node, often in an opaque way.

Deeper work in the direction of modelling query languages for the Web also exists, e.g., in [72] where a UML modelling of the language Xcerpt is shown.

3.6.3 Tests in ECA Rules for the Web

Tests can be expressed formulas in any logic, e.g., First-Order logic (where an XML markup is given in [11]), F-Logic, XPath-Logic, or XQuery or Xcerpt (note that in these languages, expressions that have an empty result also yield the truth value "false").

3.6.4 The Action Component

Composite actions can e.g. be described by *process algebras* like CCS or CSP (which makes also model checking available for verification). The composers here are e.g. (in CCS) + denoting alternative composition, $\cdot$ denoting sequential composition, $\times$ denoting parallel processing, and a fixpoint (iteration) composer. The markup of such terms is again straightforward. Variables that are communicated to the action component are accessed as described above for the event component (and also the query component).
Additional semantics depends on the action language. Possible features are

- iteration over (collection- or set-valued) variables,
- binding and quantification of local variables,
- transactions, etc.

The action language is the most independent one since it only receives variable bindings (which can be mapped by a wrapper to any language interface), but does not return anything. The structure and markup of the action component becomes relevant when reasoning about a system should be done.

Existing (sub)languages.

Again, existing languages and tools can be employed in a service-oriented architecture as described in Section 3.7:

- complex updates as provided by XChange,
- calling arbitrary Web Services (by sending SOAP messages).

3.7 Architecture: Languages and Processors as Resources

Rules on the semantic level, i.e., RDF or OWL, lift ECA functionality wrt. two (independent) aspects: first, the events, conditions and actions refer to the ontology level as described above. On an even higher level, The above rule ontology and event, condition, and action subontologies regards rules themselves as objects of the Semantic Web. Together together with the languages and their processors, this leads directly to a resource-based approach: every rule, rule component, event, subevent etc. becomes a resource, which is related to a language which in turn is related to other resources.

In this section, we propose a Web-Service-based architecture where each language is associated with a Web Service that implements the language. For being integrated into the ECA framework, the Web services must only implement the communication by variable bindings as discussed in Section 3.4.

3.7.1 Rules and Rule Components as Resources

Every rule is then interpreted as a network of RDF resources of the contributing ontologies (ECA, event algebras, applications etc.). Figure 3.14 shows the rule and the event component given in Example 3.14, combining two application-independent language ontologies:

- the ECA ontology (gray, doublelined),
- the SNOOP ontology of the event algebra: there, the semantics of the SNOOP operators must be available (gray).

and two application-dependent ontologies:

- the banking application-level ontology: there, the semantics of the atomic events defined in this ontology must be available (diagonally crosshatched).
- the temporal ontology: there, information about temporal events is available (crosshatched).

This point of view leads to the above service-oriented distributed architecture by associating the "responsibility" for handling the respective resources by appropriate services. Additionally, e.g. collections of (sub)events as well as complete (application-specific) rule bases can be designed, published by associating them with a URI, and reused.

A Modified Rule using a Derived Event. The banking ontology could define a derived booking event as the disjunction of debit and deposit. The rule could then directly use this derived event. In the ontology diagram, the only difference would be that the booking node would be //banking/events#booking and appear diagonally crosshatched (and its semantical information must be kept at the banking resource – but in this case it can also be used in specifications that do not know the SNOOP language).

3.7.2 Languages as Resources

The languages of the different types shown in Figure 3.9 (in the above example: SNOOP, banking, and temporal domain) are associated with resources by associating them with namespaces. The current W3C proposals do not specify what is actually behind these URIs. The following resources about languages would e.g. be reasonable:

- XML world (e.g., for languages like XHTML, GML (a Markup language for Geography), or the Mondial language): a DTD or XML Schema.
- programming languages marked up in XML (e.g., XSLT): a processor that interprets the language (e.g., a Web Service where one can send an XSLT document and an XML instance and gets back the result).
- Markup languages for application domains (e.g., Mondial, travelling, or banking): an RDF/RDFS or OWL description about the notions of that language. For languages that include events and action, these should also be specified there.
- Languages of application domains (e.g., travelling, or banking): specifications and services that provide (static and dynamic) information, e.g., portals, information brokers and event brokers.

For the basic implementation of the framework, languages must be associated with further resources of the respective ontologies that are used when actually processing rules. Depending on the type of ontology (e.g., language or application), the resources must provide the following functionality:

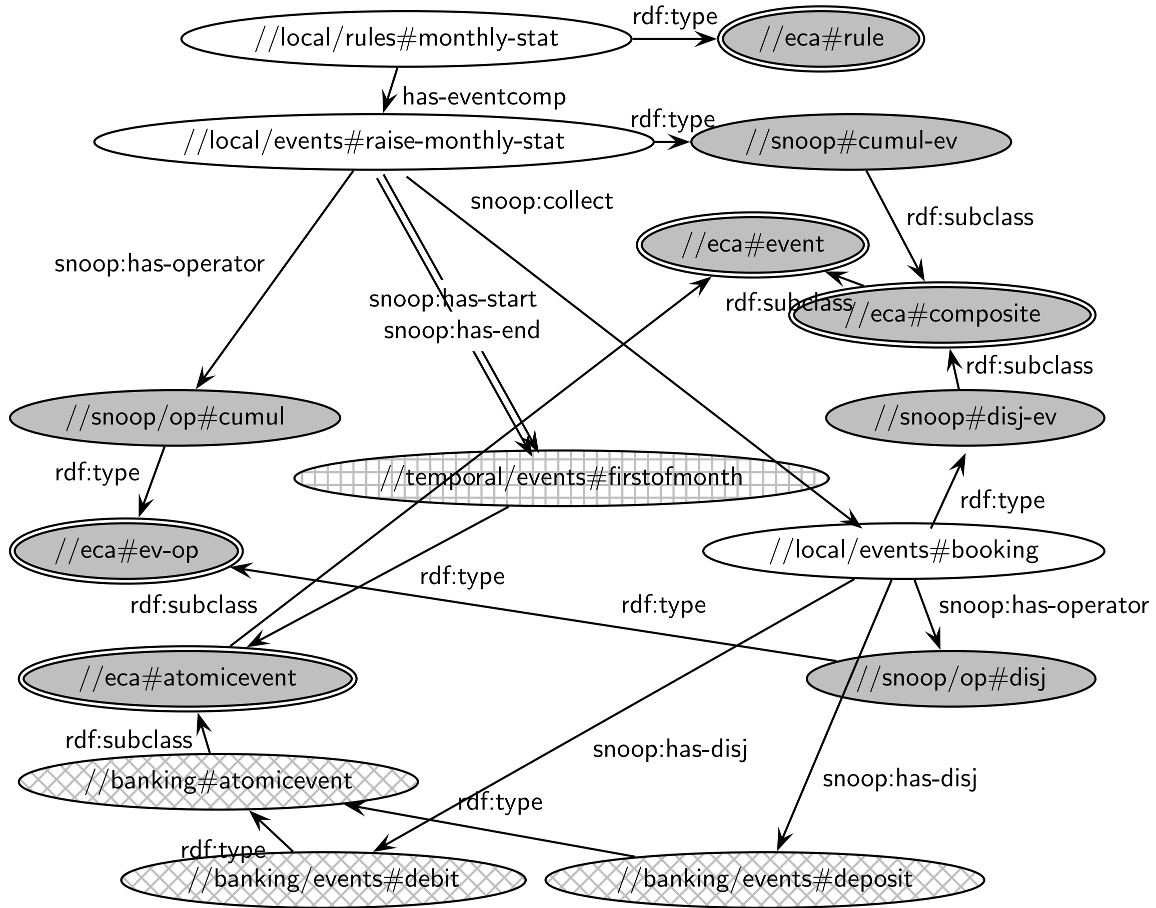


Figure 3.14: Example Rule and Event Component as Resources

- For composite event languages: a service where the composite event can be registered, and that, when informed about relevant events, runs the detection and informs the client about success (transferring the resulting variable bindings). Another possibility is a downloadable class that implements the detection algorithm and provides a suitable interface. Optionally, a formal semantics could be given using a meta-formalism that allows to *derive* a detection algorithm and to reason about.
- For queries, often the respective domain services provide also querying functionality. Alternatively, standalone query services (e.g., a saxon-based service for evaluation of XQuery queries on the Web) can be used, or queries that are needed frequently can even be registered by a *continuous query service*.
- Analogously for action languages. Often (at least in the early stage) this will be simple updates or Web Service calls.

The application domains are supported by suitable domain services that can be queried, that raise events and that are able to execute actions.

- In many cases (e.g., banking domain) the "client" knows which "server" (the bank where the account is located) provides the relevant atomic events, and where he wants the actions to be executed.
- For some application areas, there are "master services" that feel "responsible" for the ontology. These should also provide notification services for atomic events. In either case, derived events (that can locally use another event algebra) have to be defined there (since their definition conceptually also belongs to the ontology, this is not surprising). In the same way, a service for execution of atomic actions and the definition of composite actions (using any action language) can be provided.
- for many application areas (such as stocks or travel) there will be multiple services that support it. Then, either the client cann choose which one is used, or the intermediate services (ECA and the respective sublanguage) know a suitable partner.
- Application-independent domains are either supported by the client nodes themselves (such as messaging), or there are dedicated services (e.g. calendar services in the temporal domain).

In the next section, two architectural models are presented how such services can communicate for implementing this framework.

3.7.3 Architecture and Processing: Cooperation between Resources

Rules can be evaluated locally at the nodes where they are stored, or they can be registered at some *rule evaluation service*. The rule evaluation engine –both local or as a service– then manages the actual handling of rules based on the language URI references. As described above, every subconcept (i.e., events, conditions, and actions) carries the information of the actual language it uses in its xmlns:*namespace* URI attribute (note that this even allows for nested use of operations of *different* event algebras). Assume the case where the language processors are available at these URIs as a Web Service. For event detection (and analogously, execution of composite actions), at least two resources (or services) must cooperate: Event detection splits into the *event algebra* part (that is detected algorithmically by a resource representing a *language* ontology, e.g., SNOOP) and the application part that provides atomic events (detected separately by a resource representing an *application* ontology). Thus, the algebra processor must be notified about the atomic actions. This can be done in several ways:

Straightforward: The "straightforward" way is that the client C organizes the communication between the event generator(s) and the event algebra processor (see Figure 3.15): C registers rules to be "supervised" at a rule execution service R. For handling the event component, R reads the language URI of the event component, and registers the event component at the appropriate event detection service S (note that a rule service that evaluates rules with events in different languages can employ several event detection mechanisms).

During runtime, the client C forwards all received events to R, that in turn forwards them to all event detection engines where it has registered event specifications for C, amongst them, S. S is "application-unaware" and just implements the semantics of the event combinators for the incoming, non-interpreted events. In case that a (composite) event is eventually detected by S, it is signalled together with its result parameters to R. R takes the variables, and evaluates the query&test (analogously, based on the respective languages), and finally executes the action (or submits the execution order to a suitable service).

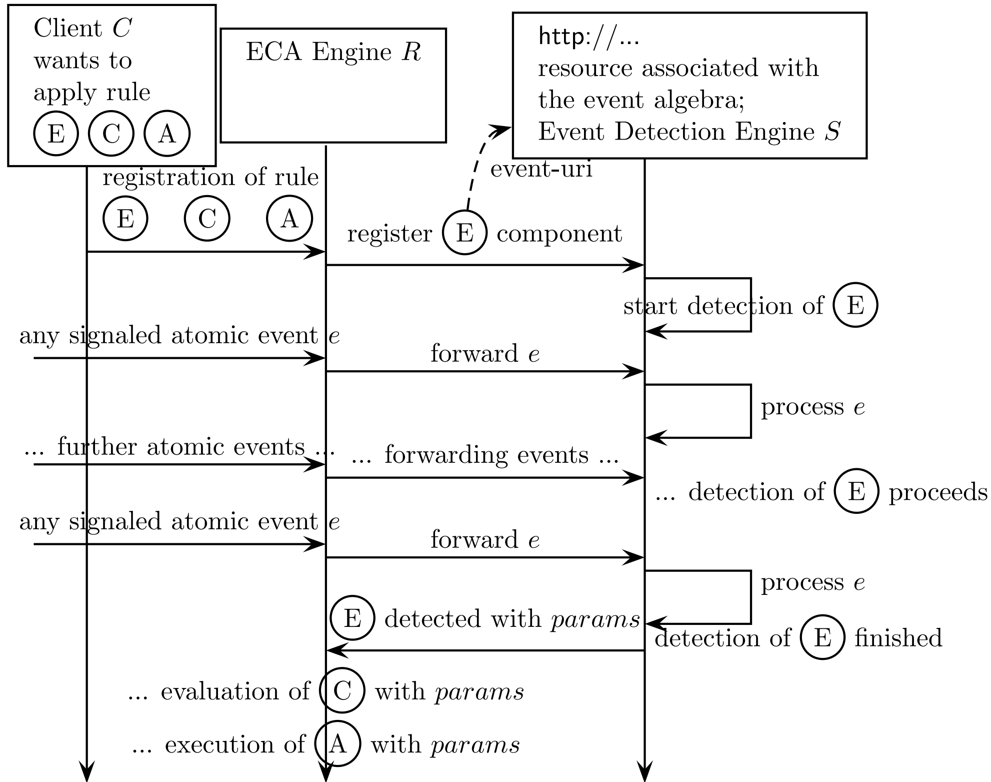


Figure 3.15: Straightforward Communication
(UML-style sequence diagram, temporal axis downwards)

Application-centered: The client submits its composite event specification to a service that is aware of all relevant events in the application domain. This service then employs an appropriate event detection service by registering the event specification, and informing it about the atomic events (e.g., "@bank: please trace the following composite event in language L on my account" (and employ a suitable event detection service for L)).

Language-centered: When a rule or an event specification is submitted for registration, this has to be accompanied by information which resource(s) provide the atomic events (e.g., "@snoop: my bank is at *uri*, please supervise my account and tell me if a composite event *ev* occurs), or the detection service even has to find appropriate event sources (by the namespaces of the atomic events). The detection service then contacts them directly. This proceeding is e.g. appropriate for booking travels where the client is in general not aware of all relevant events (e.g., "@snoop: you know better than me who is well-informed about events relevant for traveling, please detect the event ev_{travel} for me"), as illustrated in Figure 3.16: A client registers a rule (in the travel domain) at R (Step 1.1). R again submits the event component to the appropriate event detection service S ((1.2), here: snoop). Snoop looks at the namespaces

of the atomic events and sees that the travel ontology is relevant. The snoop service contacts a travel event broker (1.3) who keeps it informed (2.2) about atomic events (e.g., happening at Lufthansa (2.1a) and SNCF (2.1b)). Only after detection of the registered composite event, S submits the result to R (3) that then evaluates the Q&C component, and probably executes some actions (4.1, 4.2).

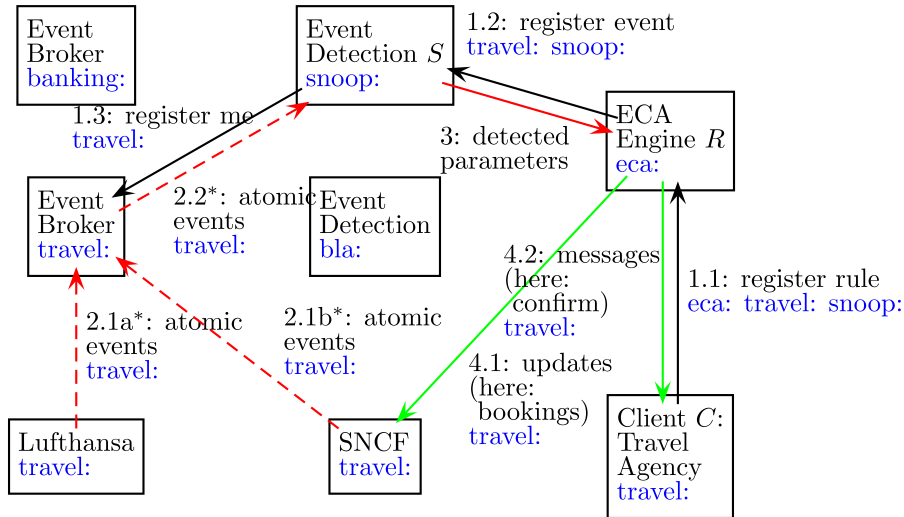


Figure 3.16: Language-Centered Communication

3.8 Normal Form, Evaluation and Optimization [Preliminary]

In Section 3.3, we have given a Normal Form for expressing ECA rules in the Semantic Web in the general case. Evaluation, optimiziations, and special cases can deviate from this normal form as long as the original semantics is preserved.

Strategic Optimization

As discussed in Section 3.5.1, there may be cases where several instances of the same event are detected simultaneously. in this case, the event detection component is allowed either to return multiple singleton answers, or a set containing multiple answers. In the latter case, the "correct" semantics would be that the ECA engine immediately splits the result set and fires one instance of the rule for each occurrence. Nevertheless, the declarative semantics allows to handle all instances in parallel, only splitting them for transactional reasons when entering the action component.

When considering to handle the query "for a set a time" or "for one instance a time",

- In case that multiple event occurrences can be handled in parallel, it is more efficient to evaluate the query part once, using free variables in all positions (or binding possible values

in an XQuery let style for restricting the result set), and afterwards *join* the result with the original tuples and then splitting the execution.

- Sometimes, it is more efficient (or even the only possible way) to evaluate the query component individually for each instance of the rule, e.g. when the "query" is actually a functional call that requires an input parameter (e.g., obtaining the rates for some hotels from a Web Service). Nevertheless, instead of splitting the complete rule, here it would be sufficient to evaluate the query for each tuple, and put the results together.

  Here, a new "style" of querying could be useful which has –up to now– only be used *inside* algebraic query evaluation: evaluate a query with an input *set of tuples of variable bindings as constraints*.

Additionally, in the Web context, if evaluation of a query for some of the values may take longer time (e.g., obtaining the rates for some hotels over the Web), separating the rule instances is strongly recommended to go ahead as soon as possible.

These issues require deeper investigation (also in combination with other aspects).

Queries vs. Conditions

The distinction between queries and conditions is often a matter of taste. The main reason to distinguish between the query and condition part were

- to allow for queries using different languages, that obtain information that is used both in the condition and the action part, and
- separating the condition.

From the declarative point of view, it is preferable to have a very strict distinction (cf. Figure 3.6). Queries only serve for *obtaining* additional information, and the conditions serve for *restricting* the execution; cf. SQL or OQL clauses of the form

```
SELECT ...
FROM (SFW-1 as A), (SFW-2 as B), ... , (SFW-n as X)
WHERE condition;
```

where in OQL, later SFW-clauses can even use the alias variables of earlier ones.

In contrast, in Logic Programming, every predicate can be seen as a subquery which, serves for obtaining additional variables, and, by using *join variables* serves for restricting the set of variable bindings. In this way, queries can also act as conditions, omitting the test component completely. At least, parts of a query+condition that are concerned with the same database can be evaluated more efficiently.

On the other hand, if the query and test components can be covered by using a single language, this can be combined in the test component. For this, the test component must interpret an empty result as "false" (which is implicit when using join variables).

Iterator-based vs. Materialized Answers

While the event component always returns a single answer that initializes the execution of a rule, queries in general return multiple answers. Depending on the implementation, these are returned all-at-a-time, or as an iterator that enumerates one answer after the other. In the latter case, the firstb answers can already bereturned before all answers are computed, allowing

for pipelining with later components. Such properties of languages (or, more exactly, individual implementations) have to be described in the metadata. In case that an answer is given as an iterator, subsequent steps can already be executed.

Algebraic Optimization

The structure and markup discussed above resulted in a normal form E-C-T-A and simple communication by tuples of variable bindings (cf. Sections 3.3.2 and 3.4). For more efficient evaluation, it is possible to deviate from this normal form, in the same way as e.g. algebraic optimization in SQL. Where in SQL, a static expression is reorganized, the evaluation of rules includes a temporal component. Here, the usual strategy of "pushing selections down" can be applied in such rules as "evaluate comditions as soon as possible":

- evaluating simple conditions concerning only with information obtained from the events already after the event detection,
- evaluating simple conditions concerning only with information obtained from the events already *during* the event detection,
- mixing query and test parts, testing as soon as the required information has been collected.

Example 3.15 *Often, events are e.g., of the form "when an account goes below zero" which includes implicitly already a condition: "if there is an update of an account where the old value was* > 0 *and the new one is* < 0*". More complex conditions would e.g. be "if there is a debit on an account that runs below zero and the person has no regular income ...". Where to draw the border between "event" and "condition"?*

The above normal form specifies the following:

- *the event component only binds variables depending on the type of the event (e.g., "debit"), the account number and the amount, but does not use any constants for comparisons ("below zero").*
- *Any check of conditions is part of the condition component.*

An obvious optimization is to check if the account goes below zero already in an on-the-fly check during the event detection phase.

Strategic Optimization

The straightforward strategy evaluates the query part for each firing of a rule. In case that different events fire instances of the same rule, or that queries are used in different rules (recall that queries and also subqueries are resources that have an identifier and can be reused), synergies between evaluation of queries for different rules and instances can be exploited.

Here, strategies investigated in the areas of *query rewriting* and *query containment*, *answer caching*, *continuous query services* etc. can be applied.

Inter-rule optimization is expected to be especially promising in the service-oriented architecture, where related tasks are concentrated on dedicated services. The rule execution, event detection, and also event and information brokering services can also make use of the above strategies.

Summary

Optimization in the context of ECA rules using different languages raises a lot of issues concerning the analysis of queries and reasoning about them both on the syntactical, algebraic and logical level, and on the semantical level.

3.9 Implementation Perspectives

As pointed out in the previous section about global architectures, the implementation of the framework consists of separate modules which can also exist in different instances, even in different implementations, sharing common interfaces as defined above. The interfaces can be refined during the development with the reference prototype.

- (language-independent) ECA engine,
- (language-independent) atomic event matching engine,
- composite event detection engines,
- query language engines, optionally directly coupled with a database,
- action language engines,
- additional services like event brokers and domain information brokers.

The prototype will be implemented incrementally. There exist already the following potential component services that can potentially be integrated.

3.9.1 ECA engine prototype

The ECA engine prototype implements the functionality described in Section 3.5: registration of rules, breaking them into parts, registering the event part at an appropriate service, receiving answers (variable bindings), invoking query services, evaluating conditions and invoking action services. Depending on the intended architecture, also event forwarding is required. A first version of the ECA engine version can thus be developed by simulating the other services.

3.9.2 Event Detection

The interface functionality can be defined already without any event algebra, just using atomic events. Services for composite event detection according to the general interfaces will be developed separately. Here, the event detection of XChange and of RuleCore (see Appendix A) are directly available in the project.

XChange. XChange is currently closely connected with the Xcerpt system. Here, a wrapper that returns variable bindings instead of immediately submitting them to Xcerpt is required. A first version can be a simple Xcerpt rule that sends a message that contains simply the XML serialization of the variable bindings. For this XChange/Xcerpt must be wrapped in a Web Service. Since XChange is still under implementation and redesign, it is appropriate to wait until a stable version is available and supported.

RuleCore. The RuleCore event detection module returns variable bindings. Wrapping it as a Web Service makes it immediately integrable.

3.9.3 Queries and Updates

Here, several alternatives already exist that only need to be wrapped according to the general interface. Here, pure language implementations for stating queries on independent XML instances have to be distinguished from XML databases. In most cases, *opaque* query components will be used since there are not yet query languages using XML-markup.

Saxon. Saxon [41] is an XQuery implementation that allows to state queries against XML sources on the Web. Wrapping it as a Web Service makes it immediately integrable. Saxon does not deal with updates.

Xcerpt. Xcerpt is the XML query language developed by REWERSE WG I4. It allows to state queries against XML sources on the Web. Wrapping it as a Web Service makes it immediately integrable. A database implementation and a lifting to RDF are planned. Xcerpt supports updates of documents via XChange's update actions. Updates are currently implemented by translating them into the definition of a view and then replacing the original document by this view. Thus, triggers on the XML level as described in Section 3.2.1 are currently not supported.

Commercial XML-enabled and XML database systems. With Oracle, IBM DB2, MS SQL Server etc., most classical relational systems habe been XML-enabled in the last years. For queries, the SQLX standard [31] is supported that embeds XPath into SQL. Updates are implemented via transformations; thus triggers on the XML level as described in Section 3.2.1 are currently not supported.

eXist. eXist [32] is an open-source XML database that runs as a Web service. For queries, XQuery is supported; also XSLT is supported for transformations. Updates are possible via XUpdate and the XQuery+Update extension in the style of [70, 44]. The REWERSE participant at Göttingen develops extensions to eXist since 2003 in another project; thus an extension with internal triggers reacting upon updates is possible.

Jena. Jena is an open-source framework that provides functionality as an RDF query language; also OWL reasoners can be integrated. Updates are possible; triggers on the RDF data level as described in Section 3.2.1 are not yet supported (but can be added since it is an open-source framework).

3.9.4 Actions

The first prototype will support the following "typical" actions that can be executed immediately by the ECA service:

- raising of events and sending them to appropriate services,
- database updates (e.g., XUpdate messages to eXist),
- Web Service calls.

For an overview of database update functionality in the potential component systems, see above. Separately, services for implementing intensional actions that are reduced via ACA rules, will be developed.

3.9.5 Requirements and Evaluation

Concerning the query and update functionality, it necessary that the underlying data can be updated *and* that triggers on the database level are available for getting a "feedback" from actions to events. High-level events can then be derived.

According to the above evaluation, we envisage to use the following components when having a basic ECA module:

- an event detection service based on XChange, RuleCore, or an own tool,
- an eXist database as underlying local database that can be updated and provides database-level triggers,
- Jena (extended with database triggers) as a similar testbed on the RDF/OWL level,
- saxon as an additional pure querying service (for remote data sources that are not updated by us).
- After consultations with the Xcerpt/XChange developers at PPSWR 2005, we came to the conclusion that after further development of the Xcerpt/XChange language family in course of REWERSE I4, Xcerpt-XML, Xcerpt-RDF and XChange will be integrated later, using the experiences collected until then. Immediate extensions and adaptations to Xcerpt/XChange, and then continuous adaptation would bind too manpower.

3.10 Relationship with Existing Languages

There are several proposals for "Reactive or ECA Languages for the Web". Most of these, e.g. see [2] for a discussion, are very restricted trigger-like languages that operate on a local XML or RDF database. They are (i) covered by our approach (at least as opaque rules, but can in general also be marked up explicitly in it), and (ii) used in our approach on the lowest level for catching data-oriented events. The existing proposals can be grouped into three classes, corresponding to the discussion in Sections 3.1.2 and 3.2:

- Local triggers where E, C, and A component use only the local database (like for SQL triggers) – these are not languages for the *Web*, but only for XML or RDF *data*,
- "Web-Level triggers" whose E-component is a data-level event in the local database, the condition component uses the local database and possibly also remote ones, and the action part can include arbitrary actions on the Web level (sending messages, SOAP),
- ECA rules where the event component uses any events that are known at the node, and the condition and action component are arbitrary.

3.10.1 Triggers on XML Data

The existing proposals for XML data inherit much from SQL database triggers. Especially, all of them react only on atomic database update events.

"Active Rules for XML", [13].

The proposal for "Active Rules for XML" [13], transfers the "trigger" idea to XML. In contrast to similar proposals that use the common plain `ON ... WHEN ...DO ...` syntax, this proposal is based on an XML markup. The E, C, and A sublanguages are fixed: The event component

allows for atomic events in the local XML database (monitoring nodes specified by XPath expressions) that provide the `$old` and `$new` values of the updated data items. The condition component then works on this information and on data from the local database, using XQuery. In the action component, SOAP methods can be invoked.

It is worth noting that the design of the condition and action component deviates from the SQL style, in the same direction as the query-test-action combination in our framework: since the action component's SOAP invocation acts remote, the introduction of a separate local query component was necessary that has been done here by extending the condition component: The condition component consists of an XQuery query that binds variables in a FOR clause, tests in the WHERE, and then (for each successful binding) "jumps" to the action component that invokes a SOAP method using the variable bindings of the condition component.

Active X-Query.

The proposal of Active XQuery [12] proposes triggers for XQuery in the style of SQL triggers:

```
CREATE TRIGGER name
[WITH PRIORITY number]
(BEFORE|AFTER) (INSERT|DELETE|REPLACE|RENAME)+ OF XPathExpression+
[FOR EACH (NODE|STATEMENT)]
[XQuery-LET-clause]
[WHEN XQuery-WHERE-clause]
DO (XQuery-UpdateOP|ExternalOp)
```

The trigger is associated with an XML resource and reacts after or before changes in the XML data, with granularities as for SQL. In the same way as in our framework, a query component that binds variables (LET clause to bind variables) has been introduced before the evaluation of the condition. The actions are XQuery-update operations, or external actions.

E-C-A for XML [5].

The proposal for an "ECA Language for XML" [5] transfers the "trigger" idea to XML, using the common

ON *event* IF *condition* DO *actions*

syntax. The E, C, and A sublanguages are fixed: The event component is restricted to insertions or deletions of nodes nodes in the local XML database (monitoring nodes specified by simple XPath expressions); changes are bound to a variable `$delta`. The condition and action components are evaluated separately for each instantiation of `$delta`. The condition allows *simple* XPath expressions (that return true/false – empty/nonempty) connected by boolean operators and cannot bind extra variables. The action component is a sequence of updates of local XML data (insertion/deletion of XML nodes).

Here, the condition component extends the SQL only slightly: the condition is not restricted to the information obtained in the events, but can use *simple* path expressions to use additional information from the local database in a restricted way. The action component is weaker (but cleaner) than in SQL since it does not allow "program fragments", but only updates.

This proposal is the most restricted one that is discussed in the comparison since it is completely restricted to the local database.

3.10.2 Triggers on RDF Data

The trigger concept from [5] has been extended to RDF data in [58]. The E, C, and A sub-languages are again fixed: events are inserting or deleting an instance of a class or a triple, or updating a triple; syntax; changes are bound to a variable `$delta`. The condition is given by RDFTL's path expressions (path expressions on RDF graphs, with filters) connected by boolean operators and cannot bind extra variables. The action component is a sequence of local updates of RDF data where updates can be inserting or deleting of a resource, or inserting, deleting or updating an arc. This proposal is also completely restricted to the local database.

3.10.3 ECA Rules on XML

The XChange language has been described in Section 2. The E, C, and A parts are also fixed, using languages that are closely related to Xcerpt [20]. Events are represented as XML instances, that can be communicated and queried (XChange event messages). Composite events are supported, using "event queries" (Xcerpt query language extended by operators similar to an event algebra). The conditions are Xcerpt queries that collect variable bindings. Actions are complex updates of Web data (XML or RDF), using an update extension of Xcerpt. The action component can be executed as transaction. The language uses logical variables to communicate values between the components.

3.10.4 ECA Rules in XML

RuleCore (cf. Appendix A) is a modular system for executing active rules. The ECA rules are marked up in XML (rCML – ruleCore Markup Language) and provide a clean dictinstion between event, conditon, and action part. The focus of ruleCore are the ECA engine itself, and its event detection component; the query and action component are relatively simple, but can be replaced by more sophisticated ones. The detection component is also extensible since new operators can be added by via appropriate classes. In this aspect, ruleCore provides already a simple form of the prospective framework.

3.10.5 Coverage of our Framework

Trigger-Style languages. The trigger-style languages discussed above could find applications as low-level rules located inside the databases

- inside the local database, and
- raising higher-level events based on XML update events.

They can be embedded into the framework either as opaque rules (which is reasonable for rules that are covered inside the database), or by defining a mapping from ECA-ML to their native languages. Since they all use primitive events, standard query languages and standard update concepts or SOAP, the embedding into the component markup is straightforward. There are no components provided by them that could be added to the service-based architecture proposed in Section 3.7.

Xcerpt and XChange. The XChange/Xcerpt ECA language consists of separate parts that provide reasonable own semantics and are based on communicating variable bindings:

- event queries, closely related to event algebras (and with the same functionality; returning a variable bindings and a sequence of events that materialized a given pattern,
- a query and test language (Xcerpt), returning variable bindings,
- combinators for composite actions.

Thus, every XChange rule can be mapped easily to the ECA-ML markup using opaque components. Furthermore, given an XML markup of the above sublanguages, rules can be marked up in XML completely. The suitablility of Xcerpt/XChange as component languages in the framework has been analyzed in Section 3.9.

RuleCore. As already mentioned above, with its modular design, ruleCore provides already a simple form of the prospective framework. On the other hand, the event detection component with the current event algebra can also be employed in an event detection node in in our framework.

Acknowledgements

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] José Júlio Alferes, Ricardo Amador, and Wolfgang May. A general language for evolution and reactivity in the semantic web. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number to appear in Lecture Notes in Computer Science. Springer, 2005.

[2] José Júlio Alferes, James Bailey, Mikael Berndtsson, François Bry, Jens Dietrich, Alexander Kozlenkov, Wolfgang May, Paula-Lavinia Pătrânjan, Alexandre Pinto, Michael Schröder, and Gerd Wagner. State-of-the-art on evolution and reactivity. Technical Report IST506779/Lisbon/I5-D1/D/PU/a1, REWERSE EU FP6 NoE, 2004.

[3] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Wolfgang May, Paula Lavinia Pătrânjan, and Michael Schröder. Use cases in evolution and reactivity. Technical Report IST506779/Lisbon/I5-D2/D/PU/a1, REWERSE EU FP6 NoE, 2005.

[4] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Flavours of XChange, a reactive, rule-based language for the (semantic) web. In *Proceedings of Intl. Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML2005)*, 2005. To appear.

[5] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for XML. In *Proceedings of the 11th International Conference on World Wide Web (WWW 2002)*, pages 486–495. ACM Press, May 2002.

[6] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Int. Conf. on Very Large Databases (VLDB)*, 2003.

[7] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0. W3C working draft, World Wide Web Consortium (W3C), November 2003. `http://www.w3.org/TR/2003/WD-xpath20-20031112/`.

[8] M. Bernauer, G. Kappel, and G. Kramler. Composite Events for XML. In *13th Int. Conf. on World Wide Web (WWW 2004)*, 2004.

[9] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite Events for XML. In *13th Int. Conf. on World Wide Web*. ACM, 2004.

[10] H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. RuleML Design. RuleML Initiative, http://www.ruleml.org/, 2002.

[11] Harold Boley, Mike Dean, Benjamin Grosof, Michael Sintek, Bruce Spencer, Said Tabet, and Gerd Wagner. FOL RuleML: The First-Order Logic Web Language. `http://www.ruleml.org/fol/`.

[12] 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.

[13] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. In *Proceedings of the 10th International Conference on World Wide Web (WWW 2001)*, pages 633–641. ACM Press, May 2001.

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

[15] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In David S. Warren, editor, *Intl. Conference on Logic Programming (ICLP)*. MIT Press, 1993.

[16] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The XML stream query processor SPEX. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*. IEEE Computer Society Press, April 2005.

[17] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems (IJSWIS)*, 1(2), 2005.

[18] François Bry, Tim Furche, and Dan Olteanu. Datenströme. *Informatik Spektrum*, 27(2):168–171, 2004.

[19] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar Types and Constraints for Web Applications. In *Proc. of 14th Int. World Wide Web Conference*, Chiba, Japan, 2005. ACM.

[20] 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.

[21] François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, June 2002. (invited article).

[22] François Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Proc. 2nd Int. Workshop "Web and Databases"*, LNCS 2593, Erfurt, Germany, October 2002. Springer-Verlag.

[23] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. Int. Conf. on Logic Programming (ICLP)*, LNCS 2401. Springer-Verlag, 2002.

[24] 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.

[25] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, September 1994.

[26] Sharma Chakravarthy and D. Mishra. SNOOP: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[27] K. R. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva. SAMOS in hindsight: experiences in building an active object-oriented DBMS. *Information Systems*, 28(5):369–392, July 2003.

[28] Klaus R. Dittrich and Stella Gatziu. *Aktive Datenbanksysteme: Konzepte und Mechanismen.* International Thomson Publishing, first edition, 1996.

[29] Document object model (DOM). `http://www.w3.org/DOM/`, 1998.

[30] Michael Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Institute for Informatics, University of Munich, Germany, 2005.

[31] 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`.

[32] eXist: an Open Source Native XML Database. `http://exist-db.org/`.

[33] Florid homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`, 1998.

[34] Charles L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[35] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In *Proceedings of the 1st International Workshop on Rules in Database Systems (RIDS 1993)*, pages 23–39. Springer-Verlag, September 1993.

[36] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS 1994)*, pages 2–9. IEEE Computer Society Press, February 1994.

[37] N. Gehani, H. V. Jagadish, and O. Smueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[38] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB 1992)*, pages 327–338. Morgan Kaufmann, August 1992.

[39] Narain H. Gehani, H.V. Jagadish, and Oded Shmueli. COMPOSE: A system for composite event specification and detection. In *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*, pages 3–15. Springer-Verlag, 1993.

[40] Eric N. Hanson and Samir Khosla. An introduction to the triggerman asynchronous trigger processor. In *Proceedings of the 3rd International Workshop on Rules in Database Systems*, number 1312 in Lecture Notes in Computer Science, pages 51–66. Springer, 1997.

[41] Michael Kay. SAXON: an XSLT processor. `http://saxon.sourceforge.net/`, 1999.

[42] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and scheme. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 134 – 146, 1989.

[43] Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-based Java-scripting for a bioinformatics semantic web. In E. Rahm, editor, *International Workshop on Data Integration in the Life Sciences - DILS*. Springer, 2004.

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

[45] Mengchi Liu, Li Lu, and Guoren Wang. A Declarative XML-RL Update Language. In *Proc. Int. Conf. on Conceptual Modeling (ER 2003)*, LNCS 2813. Springer-Verlag, 2003.

[46] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1993.

[47] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalised event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

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

[49] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the semantic web: Dealing with language heterogeneity. In *Rule Markup Languages (RuleML)*, number to appear in Lecture Notes in Computer Science. Springer, 2005.

[50] Wolfgang May, José Júlio Alferes, and Ricardo Amador. An ontology- and resources-based approach to evolution and reactivity in the semantic web. In *Ontologies, Databases and Semantics (ODBASE)*, number to appear in Lecture Notes in Computer Science. Springer, 2005.

[51] Wolfgang May, José Júlio Alferes, and François Bry. Towards generic query, update, and event languages for the Semantic Web. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 3208 in Lecture Notes in Computer Science, pages 19–33. Springer, 2004.

[52] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[53] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1):1–10, 2001.

[54] Peter Naur. Revised Report an the Algorithmic Language ALGOL60. In *Communications of the ACM, Vol.3, No.5*, May 1960.

[55] Dan Olteanu. *Evaluation of XPath Queries against XML Streams*. Ph.D. thesis (Doktorarbeit), Institute of Informatics, University of Munich, 2005.

[56] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Intl. Conference on Data Engineering (ICDE)*, pages 251–260, 1995.

[57] 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.

[58] 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.

[59] N. W. Paton, editor. *Active Rules in Database Systems*. Monographs in Computer Science. Springer, 1999.

[60] Norman W. Paton. *Active Rules in Database Systems*. Springer, 1999.

[61] Paula-Lavinia Pătrânjan. *XChange: A Rule-Based Language for Reactivity on the Web*. submitted Ph.D. thesis, currently under review, Institute for Informatics, University of Munich, 2005.

[62] The ruleCore® system — advanced business situation detection. `http://www.rulecore.com`.

[63] Rule markup language (ruleml). `http://www.ruleml.org/`.

[64] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Ph.D. thesis (Doktorarbeit), Institute for Informatics, University of Munich, 2004.

[65] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Int. Conf. Extreme Markup Languages*, Montreal, Quebec, Canada, 2004.

[66] Sebastian Schaffert, François Bry, and Tim Furche. Initial draft of a possible declarative semantics for the language. Technical Report IST506779/Munich/I4-D4/D/PU/a1, REWERSE EU FP6 NoE, 2005.

[67] Scarlet Schwiderski. *Monitoring the Behaviour of Distributed Systems*. Dissertation, University of Cambridge, United Kingdom, April 1996.

[68] Software AG. Tamino – an internet database system, 2001. `http://www.tamino.com/`.

[69] Ryan Stansifer. *Theorie und Entwicklung von Programmiersprachen*. Prentice-Hall, 1995.

[70] 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.

[71] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.

[72] G. Wagner, C. V. Damásio, and S. Lukichev. First-version rule markup languages. Technical Report IST506779/Eindhoven/I1-D3/D/PU/ab1, REWERSE EU FP6 NoE, 2005.

[73] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[74] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[75] Niklaus Wirth. *Extended Backus-Naur Form (EBNF)*, 1996. ISO/IEC 14977:1996(E).

[76] World Wide Web Consortium (W3C), http://www.w3.org/TR/soap. *Simple Object Access Protocol (SOAP) 1.1*, 2000.

[77] XML:DB Initiative, http://www.xmldb.org/xupdate/. *XUpdate - XML Update Language*, September 2000.

[78] XML Syntax for XQuery 1.0 (XQueryX). `http://www.w3.org/TR/xqueryx`, 2001.

[79] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.

Appendix A

ruleCore and the ruleCore Markup Language (rCML)

In this chapter, that has been done in collaboration with Marco Seiriö (marco@analog.se) from *Analog Software*, we present an overview of ruleCore[1] [62] and the ruleCore Markup Language (rCML)[2], a language that is used for specification of events and rules in ruleCore. Moreover, for illustration purposes, we show how to encode in rCML a selection of use cases from deliverable [3] that were already presented above for XChange. We have also experimented with ruleCore and rCML for implementing *Use Case 6.4.2 (Mirroring, Actuality, and Consistency of data in SCOP and PDB)*, i.e., a bioinformatics use case previously reported in [3]. A first implementation of the bioinformatics use case is expected to be available later this year, and is not further detailed in this report.

A.1 A brief overview of ruleCore

In this Section we present an overview of the rule engine ruleCore [62]. RuleCore is an active middleware implemented in Python, Qt, XML, and it supports ECA rules and event monitoring in heterogeneous environments. For example, a broker system can be used to integrate heterogeneous systems, and ruleCore can be attached to such a broker system and react to events that are sent through the broker.

A.1.1 Architecture

The ruleCore engine is built around a concept of loosely coupled components and is internally event driven. Components communicate indirectly using events and the publish/subscribe event passing model. The functionality of the ECA rules are provided by a number of components working in concert, where each component provides the functionality in a well defined small area. As the components are not aware of the recipient of the event they publish, it is easy to reconfigure the engine to experiment with other models besides the more well known ECA model. For example, one could insert an additional processing step between any of the event,

[1]ruleCore is a registered trademark of MS Analog Software kb
[2]rCML is a trademark of MS Analog Software kb

condition or action steps. All internal and external events are stored in a relational database (PostgreSQL). Storing the event occurrences in a database implies that traditional database tools can be used for off-line analysis, visualization, simulation and reporting.

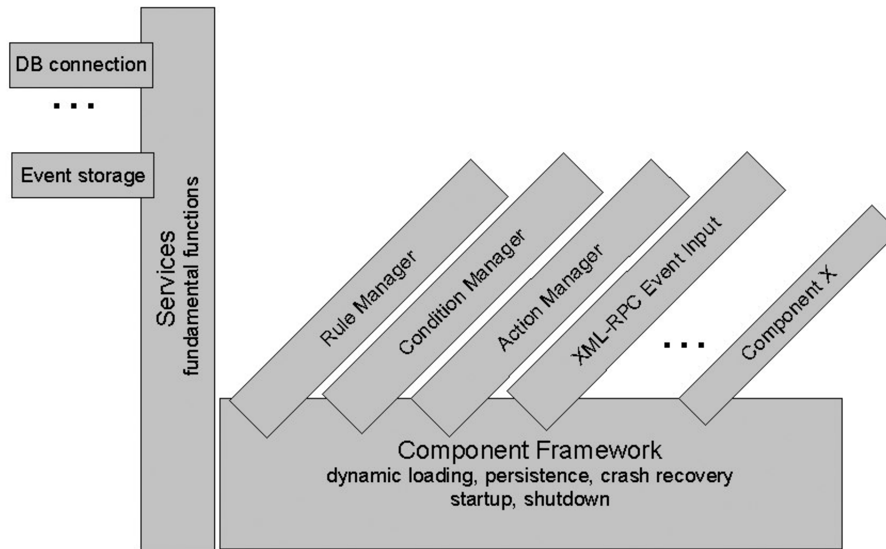


Figure A.1: ruleCore architecture

At the core of ruleCore lies a component framework, see Figure A.1. The framework provides services for loading, initializing, starting and stopping components. It also handles persistence for the components and manages automatic crash recovery for the engine.

Components that receive events contain a local worker thread that processes incoming events. Processing of events are done asynchronously and in parallel in the different components. The framework controls the starting and stopping of threads in order to provide for an ordered and consistent startup and shutdown procedure. A service layer provides simple services that are available for all components and the component framework itself. The services of the service layer are always accessible as opposed to the services provided by the components that must be loaded and initialized before usage. The service layer also provides for encapsulation of external components such as databases.

The components that provide the functionality can broadly be divided into three groups.

- *Input components* are responsible for implementing support for a specific transport protocol and accepting events through it. Currently support exists for receiving events with XML-RPC, TCP/IP Sockets, SOAP, IBM WebSphere MQ, and TIBCO Rendezvous.
- *Rule components* provide the functionality of the rules. The rule manager, condition manager and action manager components work together using event passing to implement the ECA rule execution model.

- *Support components* provide functionality that is directly or indirectly used by other components. In this group we find components for event routing, event flow management, persistent state management and management of the configuration of the engine.

A.1.2 Situation detector

In ruleCore terminology a composite event is called a *situation*. As the main focus of ruleCore is situation detection, we describe the functionality of the situation detection component in more detail.

The situation detector is implemented by using a number of detector nodes connected in a tree structure called a detector tree (similarly to what have been described above for XChange). Each ECA rule instance contains its own private instance of a detector tree, where each node in the detector tree implements some type of event detection.

Each node in the detector tree decides locally what event(s) to subscribe to, for example, a node might need to know when a specific point in time occurs and can then subscribe to a timer event in order to be informed when this particular point in time occurs. When a node detects a change it considers to be of interest it sends an event to its parent node.

When the root node in the detector tree receives an event signal, the situation is considered detected and the rule instance is triggered for condition evaluation. The situation detector can also detect if there is no possibility for detecting the situation in the future and will inform its enclosing rule instance about this fact which will then delete itself.

A.2 The ruleCore Markup Language (rCML)

In this Section we describe the ruleCore Markup Language (rCML) that is used for specification of events and rules in ruleCore. All described features of rCML have been implemented and are supported by ruleCore. Encodings in rCML are in UTF-8. In order to ease comparisons with the description of the general model above, in this section we also use the term composite event rather than situation, as usually coined in ruleCore.

A.2.1 Specification of Event Types

Events are specified inside the <event-defs> element and each individual event type is described with an <event-def> element. Two different event types are supported in rCML: basic events and composite events.

A.2.1.1 Basic Events

A basic event in rCML is defined with an <event-def> element that has two attributes:

- **type**. The attribute *type* is set to *basic* for basic events.
- **name**. A unique value for the *name* attribute that identifies the event.

Event parameters are specified inside the sub element <parameters> and each individual parameter is specified with a <parameter> element. Each <parameter> element has two attributes:

- **name**. A unique value for the *name* attribute that identifies the parameter.

- **type**. Specification of data type for the parameter. Three data types are supported:
  1. *string*. A string in valid Unicode.
  2. *number*. A decimal number according to ANSI standard X3.274-1996.
  3. *date*. A date in ISO format, e.g., YYYY-MM-DD HH:MM:SS.

Below is an example of a basic event E1 with three parameters:

```
<event-defs>
    <event-def type='basic' name='E1'>
        <parameters>
            <parameter type='string' name='parameter1'/>
            <parameter type='number' name='parameter2'/>
            <parameter type='date' name='parameter3'/>
        </parameters>
    </event-def>
</event-defs>
```

A.2.1.2 Composite Events

A simple composite event can be defined by using two basic events. However, more complex composite events are defined by building composite events out of other composite events. A composite event that contributes to the detection of another composite event is called a sub-composite event.

A composite event in rCML is defined with an <event-def> element that has two attributes:

- **type**. The attribute *type* is set to *composite* for composite events.
- **name**. A unique value for the *name* attribute that identifies the event.

Each composite event defined in rCML has four sub elements:

- <detect-event>. The <detect-event> element specifies the event that is generated when the composite event is detected.
- <no-detect-event>. The <no-detect-event> element specifies the event that is generated when the composite event can never be detected. When a composite event cannot be detected, e.g., a composite event using a time point that has already passed, the rule instance is automatically deleted. The <no-detect-event> event is generated just prior to deletion of the rule instance containing the composite event that never be detected.
- <event-selector> The <event-selector> element specifies a logical condition (or filter) for the composite event (see section on specification of conditions for some further details).
- <detector>. The composite event detector itself is defined under the <detector> element. The detector consists of a number of sub elements (event operators) that describe the composite event.

Each sub element to the <detector> element is an event operator. The following event operators, some very similar to the ones illustrated in the general framework, are supported:

- The conjunction, disjunction and sequence operators are supported, respectively, by the <and>, <or> and <sequence> elements. Similar event operators are supported in SNOOP

[25], Ode [37], and SAMOS [27], and illustrated in the general framework above. For instance, the following example specifies that event E1 is followed by event E2, and that event E3 occurs after E2:

```
<detector>
    <sequence>
        <event-ref type=’event’>E1</event-ref>
        <event-ref type=’event’>E2</event-ref>
        <event-ref type=’event’>E3</event-ref>
    </sequence>
</detector>
```

- The prior sequence operator is supported by the <prior> element. The <prior> element behaves like the <sequence> element when all of its sub events are basic events. However, when the sub events are composite events the semantics of the composite event detection are as follow. The terminating event in a sub-composite event must occur before the terminating event in the following sub-composite event occurs. The semantics of the prior sequence operator in rCML is similar to the semantics of the prior event operator as defined in Ode [37]. The following example specifies that the detection of the sub-composite event CE1 must be completed before the terminating event in the following sub-composite event CE2 occurs:

```
<detector>
    <prior>
        <event-ref type=’event’>CE1</event-ref>
        <event-ref type=’event’>CE2</event-ref>
    </prior>
</detector>
```

- The relative sequence operator is supported by the <relative> element. The <relative> element behaves like the <prior> and <sequence> element when all of its sub events are basic events. The <relative> element requires that the terminating event in a sub-composite event is detected before the detection of the initiating event in the following sub-composite event. The semantics of the relative sequence operator in rCML is similar to the semantics of the relative event operator as defined in Ode [37]. The following example specifies that the terminating event of the sub-composite event CE1 must be detected before the detection of the initiating event in the following sub-composite event CE2.

```
<detector>
    <relative>
        <event-ref type=’event’>CE1</event-ref>
        <event-ref type=’event’>CE2</event-ref>
    </relative>
</detector>
```

- The any operator is supported by the <any> element. A similar event operator is found in SNOOP [25]. The any event operator will detect when any m events out of the n specified sub events have occurred, where $m \leq n$. The order of detection of the sub events is not important. Thus, the semantics of the <any> element is similar to the <and> element, but with the difference that the user can choose that only a limited number of the sub events need to be detected. The following example specifies that the composite event is detected

when two out of the three specified sub events E1, E2, or E3 have occurred.

```
<detector>
    <any number=’2’>
        <event-ref type=’event’>E1</event-ref>
        <event-ref type=’event’>E2</event-ref>
        <event-ref type=’event’>E3</event-ref>
    </any>
</detector>
```

- The between operator is supported by the <between> element. The between event operator uses one initiating event and one terminating event to detect the composite event. Any number of events can occur between the initiating event and the terminating event. All the events that occur between the initiating event and the terminating event can be stored for condition evaluation. The between event operator is usable when the initiating and terminating events of a composite event are known but not how many events that will occur in between them. The following example specifies that the composite event is detected when E1 is followed by zero or more events before event E2 occurs.

```
<detector>
    <between>
        <event-ref type=’event’>E1</event-ref>
        <event-ref type=’event’>E2</event-ref>
    </between>
</detector>
```

- The not operator is supported by the <not> element. The classical semantics of the NOT operator when specifying composite events for ECA rules are that an event E is not detected during an interval specified by two events. For example, a composite event NOT E3 (E1,E2) is detected if event E3 is not detected between the detection of E1 and E2. Previous systems [24, 27] have restricted the use of the NOT operator to: (i) a conjunction [24], i.e., event E3 should not occur between (E1 and E2), or ii) a time interval [27], i.e., event E3 should not occur between 18:00 and 20:00.

  The approach taken in rCML generalize the usage of the NOT operator to any type of event interval. Thus, the NOT operator extends previous usage of the NOT operator for specifying composite events for ECA rules. The following example specifies that the composite event is detected when event E5 is not detected during the detection of the sequence (E1, E2,E3).

```
<detector>
    <sequence>
        <event-ref type=’event’>E1</event-ref>
        <event-ref type=’event’>E2</event-ref>
        <event-ref type=’event’>E3</event-ref>
        <not>
            <event-ref type=’event’>E5</event-ref>
        </not>
    </sequence>
</detector>
```

- The count operator is supported by the <count> element. The count event operator, which

is a simple form of cumulative operator, is used to count how many times its only sub event is detected within an interval. The interval is configured in such a way that the count operator knows when it should start and stop counting event occurrences. Thus, a <count> element is either in an open state (counting) or in a closed state (not counting). The <countcfg> element has six sub elements that must be in the cocfg namespace:

1. <open-output> - This element specifies the value a <count> element has when it is in the open state. Possible values are:
   - true - Output true to the <count> element.
   - false - Output false to the <count> element.
   - input - Output the same value as the <count> element receives from the its sub event.
2. <closed-output> - This element specifies the value a <count> element has when it is in the closed state. The possible values are the same as for <open-output>.
3. <open-count> - This element describes when to activate (or open) the count operator. It contains an integer that specifies the number of event occurrences that must have occurred before the counter starts.
4. <close-count> - This element describes when to close the count operator. It contains an integer that specifies the number of event occurrences that must have occurred before the counter is closed. The <close-count> integer should be greater than the <open-count> integer.
5. <periodic> - This element is used to specify a periodic behaviour of the <count> element. It means that the <count> element is in the open state as many times as specified by <open-count> and then in the closed state as many times as specified by <close-count>. Valid values are "True" or "False".
6. <initial-state> - This element specifies the initial state of the <count> element, which can be *Open* or *Closed*.

In its simplest form, the count operator counts the number of event occurrences:

```
<count>
    <event-ref>E1</event-ref>
</count>
```

In more advanced forms, the count operator counts the number of event occurrences when specific conditions are met. For example, the following composite event ignores the first two occurrences of E1, and counts the following three occurrences of E1:

```
<detector>
    <count>
        <cocfg:countcfg xmlns:cocfg='http://www.rulecore.com/cocfg'>
            <cocfg:open-output>true</cocfg:open-output>
            <cocfg:closed-output>false</cocfg:closed-output>
            <cocfg:open-count>2</cocfg:open-count>
            <cocfg:close-count>5</cocfg:close-count>
            <cocfg:periodic>False</cocfg:periodic>
            <cocfg:initial-state>Open</cocfg:initial-state>
        </cocfg:countcfg>
```

```
        <event-ref type=’event’>E1</event-ref>
    </count>
</detector>
```

- The timeport operator is supported by the <timeport> element, and it supports specification of absolute, relative, and periodic time events. Similar events have been proposed by the active database community.

  The following example in rCML specifies an absolute time event Event2, i.e., the timeport is opened 15th of June 2004 at 12:30:30, and that the timeport closes 15th of June 2004 at 12:30:30.

```
<detector>
    <timeport>
        <tpcfg:timers xmlns:tpcfg=’http://www.rulecore.com/tpcfg’>
            <tpcfg:timer tpcfg:name=’Timer’>
                <tpcfg:start-date>
                    <tpcfg:year tpcfg:mode=’constant’>2004</tpcfg:year>
                    <tpcfg:month tpcfg:mode=’constant’>6</tpcfg:month>
                    <tpcfg:day tpcfg:mode=’constant’>15</tpcfg:day>
                    <tpcfg:weekday tpcfg:mode=”> </tpcfg:weekday>
                    <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                    <tpcfg:minute tpcfg:mode=’constant’>30</tpcfg:minute>
                    <tpcfg:second tpcfg:mode=’constant’>30</tpcfg:second>
                </tpcfg:start-date>
                <tpcfg:stop-date>
                    <tpcfg:year tpcfg:mode=’constant’>2004</tpcfg:year>
                    <tpcfg:month tpcfg:mode=’constant’>6</tpcfg:month>
                    <tpcfg:day tpcfg:mode=’constant’>15</tpcfg:day>
                    <tpcfg:weekday tpcfg:mode=”> </tpcfg:weekday>
                    <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                    <tpcfg:minute tpcfg:mode=’constant’>30</tpcfg:minute>
                    <tpcfg:second tpcfg:mode=’constant’>30</tpcfg:second>
                </tpcfg:stop-date>
                <tpcfg:open-output>true</tpcfg:open-output>
                <tpcfg:closed-output>false</tpcfg:closed-output>
            </tpcfg:timer>
        </tpcfg:timers>
        <event-ref alias=’e2,alias2’ type=’alias’>Event2</event-ref>
    </timeport>
</detector>
```

  The output of a <timeport> element is controlled by a timer. The timer is configured to open and close the <timeport> element at specific dates and times. Each date specification consists of seven fields: year, month, day, weekday, hour, minute, and second. Each of the seven fields can be individually set to different modes that decides how the actual value of the time field will be calculated:

  – The *constant* mode is the simplest. It requires a constant to be entered for the date part,

  such as 2004 for the year. The constant mode requires you to know in advance the exact date or time for the field.

  - In order to base dates on the occurrence of events, the *offset* mode can be used. The offset mode lets you specify an offset in relation to an event occurrence. This is useful if a date or time should occur some time after a certain event. To create a time that occurs ten minutes after a certain event you would set all the fields modes to offset and set the value to zero for all fields except the minutes that would be set to +10.
  - Dates and times that occur at a regular interval can be specified with by using the *each* mode of a field. For example, if the hour field is set to mode *each*, the time will occur each hour. By setting other fields to constant mode the time will occur only on those times and dates. This could for example be used to have a time to occur each day but only during a certain month.

  The day and weekday fields have a special relationship. The weekday mode of the day field can be used to set the date to a specific weekday of the month. For instance it could be used to set the day to the second Tuesday of the month. By using negative offsets the day is calculated from the end of month. This makes it easy to specify, for example, the last Sunday of the month. The constant, offset and each modes of the day field works as for the other fields.

  If the weekday field is used, it will be processed as the last step in the date calculation to move the day of the week to the specified weekday. This enables for example easy specification of dates like next Friday. This would be done by setting the day field to offset mode with value +6 and at the same time setting the weekday field to Friday.

  Each time field does not have to have the same mode. By using different combinations of the modes for the different part it is possible to specify dates in an advanced way, for example, specification of a time point that occurs five hour after an certain event but on the last Sunday of the next month.

- The state gate operator is supported by the <state-gate> element. A <state-gate> element can be used to detect whether an object is in a particular state, for example, between 12:00 and 13:00 the object is in the "LUNCH" state. A <state-gate> element can be opened or closed depending upon whether a state exists and what parameters a state instance has. States are specified by the aid of conditions, this means that the actual specification of when the state LUNCH begins and ends is done in the <condition-def> element, i.e.. the reference <sgcfg:condition>condition12</sgcfg:condition>.

```
<detector>
    <state-gate>
        <sgcfg:state-gate-cfg xmlns:sgcfg='http://www.rulecore.com/sgcfg'>
            <sgcfg:open-output>true< /sgcfg:open-output>
            <sgcfg:closed-output>true< /sgcfg:closed-output>
            <sgcfg:state-exists>open</sgcfg:state-exists>
            <sgcfg:state-selector>
                <sgcfg:condition>condition12</sgcfg:condition>
            </sgcfg:state-selector>
        </sgcfg:state-gate-cfg>
    </state-gate>
</detector>
```

All state instances are created and deleted by a rule actions.

A.2.2 Specification of Conditions

Conditions are used in a number of places, e.g., as logical conditions for events (event selector), or as rule conditions. All types of conditions are defined under the <condition-defs> element and each individual condition is specified with a <condition-def> element.

The <condition-def> element has the following attributes:

- name - The name of the condition. All condition names must be unique.
- composite - Specifies whether the condition is a basic condition or a composite condition (contains logical operators). Allowed values are “yes” and “no”
- always-true - Allowed values are “yes” and “no”

Each condition definition starts with a parameter selection defined with a number of elements <parameter> under the <parameters> element. The parameter selection selects parameters that should be used in the condition evaluation from a number of events. The parameter selection contains the name of the event, defined with <event-ref> element. The <param-ref> element defines which parameter of the event should be selected. A selection of event instances can be done with the <instance> element. Valid values for the <instance> element are ’first’, ’last’, ’all’, an integer selecting the n:th event, or an range of events specified with two integers, for example ’2-5’. The <function> element defines a function that should be applied to the selected parameter before is it selected for evaluation in the condition.

```
<condition-defs>
    <condition-def always-true=”no” composite=”yes” name=”Condition 1”>
        <parameters>
            <parameter name=”selection 1”>
                <event-ref>Event1</event-ref>
                <param-ref>parameter 1</param-ref>
                <instance>first</instance>
                <function> mul(, 1.3)</function>
            </parameter>
        </parameters>
    <condition-def>
</condition-defs>
```

The <expression> element specifies a number of expressions under <expressions>. Each expression selects two parameter selections, one left hand side (lhs) and one right hand side (rhs). These are evaluated using the operator defined in the <operator> element.

```
<expressions>
    <expression name=”c1”>
        <lhs>
            <param-ref>selection 1</param-ref>
        </lhs>
        <operator>equal</operator>
```

```
        <rhs>
            <value>42</value>
        </rhs>
    </expression>
</expressions>
```

A number of expressions can be combined with the logical operators and, or and not using the <composite-condition> element. The <condition-ref> element is used to refer to expressions created with the <expression> element. The <composite-condition> is used to combine several <expression> elements with the logical operators And, Or and Not.

```
<composite-condition>
    <or>
        <condition-ref>c1< /condition-ref>
        <condition-ref>c2</condition-ref>
    </or>
</composite-condition>
```

A.2.3 Specification of Actions

Actions are defined under the <action-defs> element and each individual rule action is specified using the <action-def> element. An <action-def> element can in turn be composed of several <action-item> elements that are executed in the order they are defined. Thus, <action-def> element can launch the execution of two applications, where each application call is defined by a separate <action-item> element.

Four types of rule actions are supported in rCML:

- script - Script actions execute external scripts or applications
- event - Event actions send event occurrences to the ruleCore rule engine
- create_state - Rule actions that create new state items
- delete_state - Rule actions that delete state items

Below is an example of a script action that executes an external application.

```
<action-defs>
    <action-def name='Action1'>
        <action-item type="script" name="exec app">c:\actions\action1.exe
        </action-item>
    </action-def>
<action-defs>
```

Below is an example of an event action that send an event occurrence to the rule engine.

```
<action-defs>
    <action-def name='Action1'>
        <action-item type='event' name=''>Event1</action-item>
    </action-def>
</action-defs>
```

Rule actions that create a new state item have the following parameters:

- single, i.e., a single state item is created. If an attempt is made to create a new state nothing happens, the creation is just ignored.
- single with replace option. Only one single state instance is active but a new state creation will replace the currently existing state.
- multiple. This parameter creates a new state instance each time the rule action is executed. If a state instance already exists the old state instance is replaced with the new one.

A.2.4 Specification of ECA Rules

ECA rules are specified inside the <rules> element and each individual ECA rule is described with a <rule> element. The <rule> element has the following attributes:

- **name**. A unique value for the name attribute that identifies the rule.
- **create**. The create attribute controls rule instance creation. Possible values are:
  1. *single* - only a single rule instance is created.
  2. *single_replace* - only a single rule instance exists at any point in time. This implies that when a new initiator event occurs, the old rule instance is deleted and replaced with a new rule instance.
  3. *init* - create a rule instance each time an initiator event occurs.
  4. *reject* - create a new rule instance if the event is rejected by all rule instances. Rejection can be done by the event selector of the rule.
- **parameter**. Controls how the event parameters of each event are stored during the event detection process. Allowed values are:
  1. *append* - store all parameters of each event that are involved in the detection of the current event,
  2. *first* - store only the parameters of the first event in each situation,
  3. *last* - store only the last parameter of the event detection of each event type, or
  4. *never* - do not store parameters at all.

The <rule> element has the following subelements:

- The <description> element contains a description of the rule. This subelement is only for user convenience and it is not used by the engine.
- The <event-ref> element contains a reference to the composite event that triggers the rule. The main target of applications for rCML are applications that react on composite events. However, a simple basic event can also act as a triggering event for a rule by constructing a composite event with only one sub event.
- The <condition-ref> contains a reference to a condition definition element <condition-def> that specifies a condition that is evaluated when the rule is triggered by its event.
- The <action-ref> contains a reference to a <action-def> element that is executed if the rule condition is evaluated to true.
- The <minus-action-ref> contains a reference to a <action-def> element that is executed if the triggering event can never be detected.

- The <instance-limit> element is used to limit the number of the rule instance for each type of rule. Possible values are:
  1. None
  2. An integer specifying the maximum number of rule instances.

The <event-ref>, <condition-ref> and <action-ref> and <minus-action-ref> all contain an attribute called *enabled* with the possible values of *yes* or *no*. Thus a rule whose rule condition should always be evaluated to true is specified as <condition-ref enabled='no'></condition-ref>.

Below is an example of an ECA rule in rCML:

```
<rules>
    <rule parameter='append' create='single' name='Rule1'>
        <description>A description of this rule</description>
        <event-ref enabled='yes'>E1</event-ref>
        <condition-ref enabled='yes'>Condition12</condition-ref>
        <action-ref enabled='yes'>Action1</action-ref>
        <minus-action-ref enabled='yes'>Action2</minus-action-ref>
        <instance-limit>None</instance-limit>
    </rule>
</rules>
```

A.3 Modelling Use Cases in rCML

All use cases in this section have been modelled in the ruleCore Designer, hence some additional (internal) tags, e.g., <scope>, are present that are not actually needed for the REWERSE scenarios. An implementation of these uses cases with XChange can be found above, in Section 2.6, where some further detail on the considered scenario are given. For a detailed specification of the scenario and the use cases see [3].

A.3.1 Use Case 4.2.1 - Changing Phone Number

Use Case 4.2.1 (Changing Phone Number) Phone numbers (or any other contact details) are updated at the participants nodes (XML format). The updates have to be propagated to the WG nodes and to the central node.

For this example, we use a local ECA rule that sends an update message to the WG nodes and the central node.

As only composite events can be directly linked to rules in rCML, a composite event changeOfPhone is created that simply detects the basic event e1 that is delivered from the event source. A wrapper needs to be added at the event source that can send the event to the ruleCore. When rule *R1* is triggered by the event *changeOfPhone* it will trigger *Action1*. The rule action is composed of two rule actions that will invoke external scripts that forward the update information to the WG nodes and the central node. All event parameters that participated in the event detection, i.e., phone and person, are also available to rule actions.

```
<rules>
    <rule create='single' parameter='append' name='R1'>
```

```
        <description>This rule is used for use case 4.2.1</description>
        <event-ref enabled=’yes’>changeOfPhone</event-ref>
        <condition-ref enabled=’no’/>
        <action-ref enabled=’yes’>Action1</action-ref>
        <minus-action-ref enabled=’no’/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type=’basic’ name=’e1’>
        <parameters>
            <parameter type=’string’ name=’phone’/>
            <parameter type=’string’ name=’person’/>
        </parameters>
    </event-def>
    <event-def type=’composite’ name=’changeOfPhone’>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=”>
            <condition/>
            <scope>global</scope>
            <events>
                <event-ref type=’type’>e1</event-ref>
            </events>
        </event-selector>
        <detector>
            <event-ref type=’event’>e1</event-ref>
        </detector>
    </event-def>
</event-defs>
<condition-defs/>
<action-defs>
    <action-def name=’Action1’>
        <action-item type=’script’ name=’send-update-to-WG-nodes’/>
        <action-item type=’script’ name=’send-update-to-central-node’/>
    </action-def>
</action-defs>
```

In summary, the above rule is an event-action rule that is triggered by a basic event, and sends two update messages in the action part.

A.3.2 Use Case 4.2.7 - Progress Reports

Use Case 4.2.7 (Progress Reports) The deadline for the progress report is inserted into the central node, and then communicated to the WGs nodes. From there, the persons are called to send input (by mail, probably 10 days before the deadline), and the coordinator is called to produce the report. The coordinator then puts the report in the WG node. An active rule then

publishes the report on the WGs Web page, and removes the deadline entry from the WG node and from the coordinators person entry in the participants node. Depending on push or pull strategy, (i) the WG node sends the report to the central node, or (ii) the central node reacts on the remote event on the WG node:

This scenario can be solved in a number of different ways. For example, the central node can simply use a basic event (reportSubmitted) and store the progress report. The disadvantage of this approach is that it is not possible to determine which WGs that have submitted their progress reports by only investigating the event. Instead, a query must be specified that investigates which reports are already submitted and stored.

Our solution is that each WG has their unique event id, e.g., *submitProgress Report-I1* that is sent to the central node. The progress report is sent to the central node as an event parameter. Rule *R2* is triggered by a disjunction *progressReportSubmitted* that collects all valid submitProgressReport-XX events and store each progress report on the central node.

Rule *R3* is triggered by a conjunction *allReportsReceived*, and the rule action sends a notification to the project officer when all progress reports have been received.

```
<rules>
    <rule create=’single’ parameter=’append’ name=’R2’>
        <description>This rule is used in use case 4.2.7 for storing progress
        reports that are submitted.</description>
        <event-ref enabled=’yes’>progressReportSubmitted< /event-ref>
        <condition-ref enabled=’no’/>
        <action-ref enabled=’yes’>storeProgressReport</action-ref>
        <minus-action-ref enabled=’no’/>
        <instance-limit/>
    </rule>
    <rule create=’single’ parameter=’append’ name=’R3’>
        <description>This rule is used in use case 4.2.7 for notifying
        the project officer when all progress reports have submitted.</description>
        <event-ref enabled=’yes’>allReportsReceived</event-ref>
        <condition-ref enabled=’no’/>
        <action-ref enabled=’yes’>notifyProjectOfficer</action-ref>
        <minus-action-ref enabled=’no’/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type=’basic’ name=’submitProgressReport-I1’>
        <parameters>
            <parameter type=’string’ name=’progressReport’/>
        </parameters>
    </event-def>
    <event-def type=’basic’ name=’submitProgressReport-I2’>
        <parameters>
            <parameter type=’string’ name=’progressReport’/>
        </parameters>
    </event-def>
```

```
    <event-def type=’composite’ name=’progressReportSubmitted’>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=’None’>
            <condition/>
            <scope>global</scope>
            <events>
                <event-ref type=’type’>submitProgressReport-I1</event-ref>
                <event-ref type=’type’>submitProgressReport-I2</event-ref>
            </events>
        </event-selector>
        <detector>
            <or>
                <event-ref type=’event’>submitProgressReport-I1</event-ref>
                <event-ref type=’event’>submitProgressReport-I2</event-ref>
            </or>
        </detector>
    </event-def>
    <event-def type=’composite’ name=’allReportsReceived’>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=”>
            <condition/>
            <scope>global</scope>
            <events>
                <event-ref type=’type’>submitProgressReport-I1</event-ref>
                <event-ref type=’type’>submitProgressReport-I2</event-ref>
            </events>
        </event-selector>
        <detector>
            <and>
                <event-ref type=’event’>submitProgressReport-I1</event-ref>
                <event-ref type=’event’>submitProgressReport-I2</event-ref>
            </and>
        </detector>
    </event-def>
</event-defs>
<condition-defs/>
<action-defs>
    <action-def name=’storeProgressReport’>
        <action-item type=’script’ name=’storeReport’/>
    </action-def>
    <action-def name=’notifyProjectOfficer’>
        <action-item type=’script’ name=’sendMailToPO’/>
    </action-def>
</action-defs>
```

A.3.3 Use Case 4.2.13 - Progress Report Late

Use Case 4.2.13 (Progress Report Late) For each progress report, a deadline is specified when it must arrive, e.g., <todo type=”progressreport” number=”1-2005” deadline=”28.2.2005”/> Then, there is a rule that states that if for any WG, the progress report has not been checked in until noon at the day of the deadline, a message is sent to the WGs coordinator.

For any WG, we first detect a sequence, e.g., *todoAnnouncement-I1* followed by *submitProgressReport-I1*. An internal composite event (conjunction) is used to detect the completion of the sequence and that the progress report has not occurred. In order for this to work we need to declare that the *submitProgressReport* event cannot be an initiator, i.e., <event-ref init=’no’ type=’event’>submitProgressReport-I2</event-ref>. In case the progress report is submitted before the deadline the detector knows that the overall composite event can never be completed and removes the rule instance.

```
<rules>
    <rule create='single' parameter='append' name='R4'>
        <description>This rule is used in use case 4.2.13 for sending
        mails to WG coordinators that have not submitted their progress
        reports on time</description>
        <event-ref enabled='yes'>lateReports</event-ref>
        <condition-ref enabled='no'/>
        <action-ref enabled='yes'>sendMailToWGCO</action-ref>
        <minus-action-ref enabled='no'/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type='basic' name='submitProgressReport-I1'>
        <parameters>
            <parameter type='string' name='progressReport'/>
        </parameters>
    </event-def>
    <event-def type='basic' name='submitProgressReport-I2'>
        <parameters>
            <parameter type='string' name='progressReport'/>
        </parameters>
    </event-def>
    <event-def type='basic' name='todoAnnouncement-I1'>
        <parameters>
            <parameter type='date' name='deadline'/>
        </parameters>
    </event-def>
    <event-def type='basic' name='todoAnnouncement-I2'>
        <parameters>
            <parameter type='date' name='deadline'/>
        </parameters>
    </event-def>
    <event-def type='composite' name='lateReports'>
```

```
<detect-event/>
<no-detect-event/>
<event-selector name=’None’>
    <condition/>
    <scope>global</scope>
    <events>
        <event-ref type=’type’>submitProgressReport-I1</event-ref>
        <event-ref type=’type’>todoAnnouncement-I1</event-ref>
        <event-ref type=’type’>submitProgressReport-I2</event-ref>
        <event-ref type=’type’>todoAnnouncement-I2</event-ref>
    </events>
</event-selector>
<detector>
    <or>
        <and>
            <sequence>
                <event-ref type=’event’>todoAnnouncement-I1</event-ref>
                <timeport>
                    <tpcfg:timers xmlns:tpcfg=’http://www.rulecore.com/tpcfg’>
                        <tpcfg:timer tpcfg:name=’Timer’>
                            <tpcfg:start-date>
                                <tpcfg:year tpcfg:mode=’constant’>2005</tpcfg:year>
                                <tpcfg:month tpcfg:mode=’constant’>2</tpcfg:month>
                                <tpcfg:day tpcfg:mode=’constant’>28</tpcfg:day>
                                <tpcfg:weekday tpcfg:mode=’none’/>
                                <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                                <tpcfg:minute tpcfg:mode=’constant’>0</tpcfg:minute>
                                <tpcfg:second tpcfg:mode=’constant’>0</tpcfg:second>
                            </tpcfg:start-date>
                            <tpcfg:stop-date>
                                <tpcfg:year tpcfg:mode=’constant’>2005</tpcfg:year>
                                <tpcfg:month tpcfg:mode=’constant’>2</tpcfg:month>
                                <tpcfg:day tpcfg:mode=’constant’>28</tpcfg:day>
                                <tpcfg:weekday tpcfg:mode=’none’/>
                                <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                                <tpcfg:minute tpcfg:mode=’constant’>0< /tpcfg:minute>
                                <tpcfg:second tpcfg:mode=’constant’>0</tpcfg:second>
                            </tpcfg:stop-date>
                            <tpcfg:open-output>true</tpcfg:open-output>
                            <tpcfg:closed-output>false</tpcfg:closed-output>
                        </tpcfg:timer>
                    </tpcfg:timers>
                </timeport>
            </sequence>
            <not>
                <event-ref init=’no’ type=’event’>submitProgressReport-I1</event-ref>
            </not>
```

```
        </and>
        <and>
            <sequence>
              <event-ref type=’event’>todoAnnouncement-I2< /event-ref>
              <timeport>
                <tpcfg:timers xmlns:tpcfg=’http://www.rulecore.com/tpcfg’>
                  <tpcfg:timer tpcfg:name=’Timer’>
                    <tpcfg:start-date>
                      <tpcfg:year tpcfg:mode=’constant’>2005</tpcfg:year>
                      <tpcfg:month tpcfg:mode=’constant’>2</tpcfg:month>
                      <tpcfg:day tpcfg:mode=’constant’>28</tpcfg:day>
                      <tpcfg:weekday tpcfg:mode=’none’/>
                      <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                      <tpcfg:minute tpcfg:mode=’constant’>0</tpcfg:minute>
                      <tpcfg:second tpcfg:mode=’constant’>0</tpcfg:second>
                    </tpcfg:start-date>
                    <tpcfg:stop-date>
                      <tpcfg:year tpcfg:mode=’constant’>2005</tpcfg:year>
                      <tpcfg:month tpcfg:mode=’constant’>2</tpcfg:month>
                      <tpcfg:day tpcfg:mode=’constant’>28< /tpcfg:day>
                      <tpcfg:weekday tpcfg:mode=’none’/>
                      <tpcfg:hour tpcfg:mode=’constant’>12</tpcfg:hour>
                      <tpcfg:minute tpcfg:mode=’constant’>0</tpcfg:minute>
                      <tpcfg:second tpcfg:mode=’constant’>0</tpcfg:second>
                    </tpcfg:stop-date>
                    <tpcfg:open-output>true</tpcfg:open-output>
                    <tpcfg:closed-output>false</tpcfg:closed-output>
                  </tpcfg:timer>
                </tpcfg:timers>
              </timeport>
          </sequence>
          <not>
            <event-ref init=’no’ type=’event’>submitProgressReport-I2</event-ref>
          </not>
        </and>
      </or>
    </detector>
  </event-def>
</event-defs> <condition-defs/>
<action-defs>
  <action-def name=’sendMailToWGCO’>
    <action-item type=’script’ name=’sendMailToPO’/>
  </action-def>
</action-defs>
```

A.3.4 Use Case 4.2.14 - Polls: Basic Rules

Use Case 4.2.14 (Polls: Basic Rules) Polls are evaluated as follows:

1. *when a poll entry is inserted, evaluate the set of recipients and send them a mail,*
2. *when a response comes in, store it in the database,*
3. *when the deadline is over (note that this is actually a sequential event: inserting a poll and the temporal event of the deadline), query the database and inform the project office about the result; in case that answers are missing, send another mail to these persons.*

Concerning 1, see we use a similar structure as in use case 4.2.1.

```
<rules>
    <rule create='single' parameter='append' name='R5'>
        <description>This rule is used in use case 4.2.14 for sending mails to
        participants of a poll.
        </description>
        <event-ref enabled='yes'>poll</event-ref>
        <condition-ref enabled='no'/>
        <action-ref enabled='yes'>sendMail< /action-ref>
        <minus-action-ref enabled='no'/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type='basic' name='insertOfPoll'>
        <parameters>
            <parameter type='string' name='poll'/>
        </parameters>
    </event-def>
    <event-def type='composite' name='poll'>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=''>
            <condition/>
            <scope>global</scope>
                <events>
                    <event-ref type='type'>insertOfPoll</event-ref>
                </events>
        </event-selector>
        <detector>
            <event-ref type='event'>insertOfPoll</event-ref>
        </detector>
    </event-def>
</event-defs>
<condition-defs/>
<action-defs>
    <action-def name='sendMail'>
        <action-item type='script' name='sendMail'/>
```

```
    </action-def>
</action-defs>
```

Concerning 2, we use the same approach as for storing incoming progress reports as seen in use case 4.2.7.

Concerning 3, we use a sequence operator as follows:

```
<rules>
    <rule create=’single’ parameter=’append’ name=’R6’>
        <description>This rule is used in use case 4.2.14 for sending results
        of a poll to the project officer and for sending reminders.</description>
        <event-ref enabled=’yes’>E12</event-ref>
        <condition-ref enabled=’no’/>
        <action-ref enabled=’yes’>A12</action-ref>
        <minus-action-ref enabled=’no’/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type=’basic’ name=’insertOfPoll’>
        <parameters>
            <parameter type=’string’ name=’poll’/>
        </parameters>
        </event-def>
    <event-def type=’composite’ name=’E12’>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=”>
            <condition/>
            <scope>global</scope>
            <events>
                <event-ref type=’type’>insertOfPoll</event-ref>
            </events>
        </event-selector>
        <detector>
            <sequence>
                <event-ref type=’event’>insertOfPoll</event-ref>
                <timeport>
                    <... spec of deadline>
                </timeport>
            </sequence>
        </detector>
    </event-def>
</event-defs>
<condition-defs/>
<action-defs>
    <action-def name=’A12’>
        <action-item type=’script’ name=’sendResultsToPO’/>
```

```
        <action-item type='script' name='sendReminder'/>
    </action-def>
</action-defs>
```

A.3.5 Use case 4.2.15 - complex events

Use Case 4.2.15 (Polls: Cumulative Events) The same activity can be carried out without actually storing answers, but handling them in a cumulative event (that collects the answers): when a poll entry is inserted, evaluate the set of recipients and send them a mail, if first, a poll is issued, and then a set of answers answer(person,answer) comes in, collect the set of persons and the set of answers until the deadline (atomic events: poll, answer, deadline). The overall detection of this composite event results in a small XML instance that contains the collected information.

For this use case, we use a composite event that use the insert of poll as the initiating event and the deadline as the terminating event. The event operator between is used to collect all events that occur between the insert of the poll and the deadline.

```
<rules>
    <rule create='single' parameter='append' name='R7'>
        <description/>
        <event-ref enabled='yes'>E13</event-ref>
        <condition-ref enabled='no'/>
        <action-ref enabled='no'/>
        <minus-action-ref enabled='no'/>
        <instance-limit/>
    </rule>
</rules>
<event-defs>
    <event-def type='basic' name='insertOfPoll'>
        <parameters>
            <parameter type='string' name='poll'/>
            <parameter type='string' name='poll-id'/>
            <parameter type='date' name='deadline'/>
            <parameter type='string' name='persons'/>
        </parameters>
    </event-def>
    <event-def type='composite' name='E13'>
        <detect-event/>
        <no-detect-event/>
        <event-selector name=''>
            <condition/>
            <scope>global</scope>
            <events/>
        </event-selector>
        <detector>
            <between>
```

```
                <event-ref type='event'>insertOfPoll</event-ref>
                    <timeport>
                        <... spec of deadline>
                    </timeport>
            </between>
        </detector>
    </event-def>
</event-defs>
```

A.4 Conclusions

In light of the general framework proposed in this deliverable, it is clear that ruleCore is quite developed in what regards the detection and processing of events. It contains basic events, which in principle can be anything that can be wrapped and send as an event signal to ruleCore, as well as composite events with a rich collection of event composition operators.

The event operators timeport and state gates are perhaps the most novel and powerful event operators in rCML. For example, in ruleCore one can create a composite event that collect all events that occur five minutes after the LUNCH state the second Monday in each month. In order to save space, the actual rCML code for this composite event is not shown. However, most of the rCML code has already been shown in the explanations of the event operators timeport, state gate, and between.

The condition part, which is optional, may be used to test additional conditions on events before performing the action. Conditions are logical expressions, that may call functions, and no support for external database queries is provided. In particular, unlike what happens with XChange, there is no support for querying XML databases in conditions. In this part all event parameters are available.

The actions may be (internal) raising of events, as existing in XChange and discussed in the general framework, or external (wrappers) actions. The external actions call external scripts or applications. Unlike XChange, there is no specific support for changing/updating Web data. Again, in the actions part all event parameters are available.

One feature that needs to be extended in ruleCore in the future is more advanced support for processing XML documents that are submitted as event parameters. As of now, ruleCore/rCML has no support for extracting specific values in such documents, something that in contrast is well developed in XChange. In addition, there is no support for using namespaces.

Using ruleCore together with other ECA systems have not been tested in practice. However, it should be possible to use ruleCore's event detection module and then simply forward the event detection signal to another ECA system, such as XChange, for condition evaluation and action execution, and this is something that will be pursued in the context of the general framework. The problem at this stage is that it will be time consuming to manually check correct usage of event parameters etc in both systems, since there is no support for name spaces in ruleCore.

Appendix B

Prova language description

B.1 Introduction

Prova [43] a Language for Rule Based Java Scripting, Information Integration, and Agent Programming, mainly contributed by a REWERSE partner (Dresden), is derived from Mandarax Java-based inference system. It extends Mandarax by providing a proper language syntax, native syntax integration with Java and agent messaging and reaction rules. Prova is suitable for use as a rules-based backbone for distributed web applications in biomedical data integration.

- The design goals of Prova
  - Marry the benefits of declarative and object-oriented programming;
  - Combine the syntaxes of Prolog and Java – ultimate logic and object-oriented languages;
  - Expose logic and agent behaviour as rules;
  - Access data sources via wrappers written in Java or command-line shells like Perl;
  - Make all Java API from available packages directly accessible from rules;
  - Run within the Java runtime;
  - Enable rapid prototyping of applications;
  - Offer a rule-based platform for distributed agent programming.

Prova is useful for data integration tasks when the following is important:

- Location transparency (local, remote, mirrors);
- Format transparency (database, RDF, XML, HTML, flat files, computation resource);
- Resilience to change (databases and web sites change often);
- Use of open source technologies;
- Understandability and modifiability by a non-IT specialist;
- Economical knowledge representation;
- Extensibility with additional functionality;
- Leveraging ontologies.

Example B.1 *(Declarative programming)*
Consider a table of interacting proteins. We wish to infer all interactions, direct or indirect.

In Prova, this can be specified as follows (`:-` is read as "if"):

Listing B.1: Prova example

```
% Facts (what we know)
interactDirect(a,b).
interactDirect(b,c).
interactDirect(c,d).

% Rules (how to derive new knowledge)
interact(X,Y):-interactDirect(X,Y).
interact(X,Z):-interactDirect(X,Y),interact(Y,Z).
```

The query `:- solve(interact(a,X)).`*, which can be read as "which proteins* `X` *interact with protein* `a`*?", will return the three answers* `X=b`*,* `X=c`*, and* `X=d`.

Thus, Prova follows classical Prolog closely by declaratively specifying relationships with facts and rules. Now let us consider an example where access to Java methods is directly integrated into rules.

Example B.2 *(Object-oriented programming)*
The code below represents a rule whose body consists of three Java method calls: the first to construct a String object, the second to append something to the string, and the third to print the string to the screen.

Listing B.2: Prova example

```
hello(Name):-
  S = java.lang.String("Hello "),
  S.append(Name),
  java.lang.System.out.println(S).
```

B.1.1 Examples solved with Prova

The Prova code very closely resembles the declarative Prolog specification. However, instead of relying on an internal knowledge base, which needs to be loaded entirely into memory, Prova can access the GO in a database, which is accessed as needed. In the same way as Prolog, Prova applies backward-chaining to evaluate queries.

The following Prova code first "imports" some utility functions, like *dbopen*, which opens a database connection. One database location is provided in line 4. The script evaluates the next three statements at once:

- First, it tries to bind a GO term accession number to a given name, in line 7. The answer is no if the database has no such label for any term. The predicate *name2term* is defined later in the code (lines 42-45). It opens a database connection to a current GeneOntology MySQL database scheme, constructs the where-clause for the SQL statements in line 44 and issues the SQL query in line 45. The values of the column *id* in the result set are bound to the variable T.
- Second, descendant concepts in the ontology are found for *small GTPase mediated signal transduction* with the predicate *isDesc*. In line 20, GO accession ids are bound to $T1$ if there is a term with the name $N1$ and it has an accession id. Line 21 binds all terms which are sub-concepts of $T1$ to $T2$. For this it uses the recursive definition of *isa* in line 29 which eventually queries the database using the predicate *isaDB* defined in line 36. In line 43 the database connection is opened. Line 44 just prepares the where-clause for the SQL statement used in *sql_select*, line 45.

- The third query is similar to the second one but excluding subclasses of *Rho protein signal transduction* in the result.

```
:-eval(consult("utils.prova")).

% Define database location
location(database,"GO","jdbc:mysql://server","guest","guest").

% Show the ID of the GO term small GTPase mediated signal transduction?
 :-solve(name2term("small GTPase mediated signal transduction",GO_ID)).

% Which small GTPase mediated signal transduction processes are listed in the GO?
:-solve(isDesc("small GTPase mediated signal transduction",N)).

% Which small GTPase mediated signal transduction processes
%apart from Rho are listed in the GO?
:-solve(isDesc2(N)).
isDesc2(N):-
  isDesc("small GTPase mediated signal transduction",N),
  not(isDesc("Rho protein signal transduction",N)).

% Defining a descendent
isDesc(N1,N2) :-
  name2term(N1,T1), % N1 has the term id T1
  isa(T2,T1),       % T2 is a T1
  term2name(T2,N2). % T2 has the term name N2

% A term T is a T
isa(T,T).

% Recursive definition of is-a:
% A term T2 is a T1 if T3 is a T1 and T2 is a T3
isa(T2,T1) :-
  isaDB(T3,T1),
  isa(T2,T3).

% This predicate is limited by the number of open connections
% allowed T2 is a T1 if there is a corresponding entry in the
% term2term table of the database
isaDB(T2,T1) :-
  dbopen("GO",DB),
  concat(["term1_id=",T1," and relationship_type_id=2"],WhereClause),
  sql_select(DB,term2term,[term2_id,T2],[where, WhereClause]).

% Given the name N, get the term id T
name2term(N,T) :-
  dbopen("GO",DB),
  concat(["name like ",N],WhereClause),
  sql_select(DB,term,[id,T],[where, WhereClause]).

% Given the term id T, get the term name N
term2name(T,N) :-
  dbopen("GO",DB),
  concat(["id=",T],WhereClause),
  sql_select(DB,term,[name,N],[where, WhereClause]).
```

The unmodified output of this is shown below. Note there are three paragraphs. One paragraph per *solve* statement.

```
GO_ID=4280

N="small GTPase mediated signal transduction"
N="Ras protein signal transduction"
N="Rho protein signal transduction"
N="Rac protein signal transduction"
N="regulation of small GTPase mediated signal transduction"
N="regulation of Rac protein signal transduction"
N="negative regulation of Rac protein signal transduction"
```

```
   N="positive regulation of Rac protein signal transduction"
11 N="regulation of Rho protein signal transduction"
   N="negative regulation of Rho protein signal transduction"
13 N="positive regulation of Rho protein signal transduction"
   N="regulation of Ras protein signal transduction"
15 N="positive regulation of Ras protein signal transduction"
   N="negative regulation of Ras protein signal transduction"
17 N="positive regulation of small GTPase mediated signal transduction"
   N="positive regulation of Rac protein signal transduction"
19 N="positive regulation of Rho protein signal transduction"
   N="positive regulation of Ras protein signal transduction"
21 N="negative regulation of small GTPase mediated signal transduction"
   N="negative regulation of Rac protein signal transduction"
23 N="negative regulation of Rho protein signal transduction"
   N="negative regulation of Ras protein signal transduction"
25
   N="small GTPase mediated signal transduction"
27 N="Ras protein signal transduction"
   N="Rac protein signal transduction"
29 N="regulation of small GTPase mediated signal transduction"
   N="regulation of Rac protein signal transduction"
31 N="negative regulation of Rac protein signal transduction"
   N="positive regulation of Rac protein signal transduction"
33 N="regulation of Rho protein signal transduction"
   N="negative regulation of Rho protein signal transduction"
35 N="positive regulation of Rho protein signal transduction"
```

Observant readers might notice that the answer to the question ”‘Which small GTPase mediated signal transduction processes apart from Rho are listed in the GO?”’ contains *(negative/positive) regulation of Rho protein signal transduction*. In this query we want the list of processes which are a subclass of *small GTPase mediated signal transduction* but excluding subclasses of *Rho protein signal transduction*. GO currently lists *regulation of Rho protein signal transduction* as *part_of* but not as *is_a Rho protein signal transduction*. However *(negative/positive) regulation of Rho protein signal transduction* are subclasses of *regulation of Rho protein signal transduction* but not of *Rho protein signal transduction*. Therefore the query is in fact answered correctly, but the GO contains an inconsistency in its use of part-of and is-a: *regulation of small GTPase mediated signal transduction* is-a *small GTPase mediated signal transduction* but not *part_of*. We summarize that GO sometimes lists *regulations* of a process as part of the general process and sometimes as being a subclass of it.

B.1.2 Prova as distributed Agent programming

B.1.2.1 Agents Architecture Prova-AA

Prova includes syntactically simple constructs allowing for sending messages via either JADE-HTTP or JMS communication protocols and for specifying sophisticated reaction rules for processing incoming messages. Due to the natural integration of Prova with Java, Prova-AA offers a syntactically economic and compact way of specifying agents behaviour while allowing for efficient Java-based extensions to improve performance of critical operations.

To access Prova-AA features from Prova, additional jar files should be available on the CLASSPATH and special command-line arguments should be included to launch the relevant communication platforms. A subdirectory prova-aa/examples of the main prova distribution directory contains two sets of examples one each working with JADE-HTTP and JMS protocols. The provided README files in the Prova webpage (www.semanticwebrules.org) explain the procedures needed to run the examples.

Java Message Service JMS is an industry standard message oriented middleware platform

that is part of Java 2 Enterprise Edition J2EE. Prova-AA uses a particular implementation of JMS called Joram[1]. This version has the advantage to be open source and mature, and furthermore, it is now part of a broad ObjectWeb initiative bringing together various middleware and J2EE technologies that will be used by major Linux vendors like RedHat and Suse. JMS in general has the advantage of being a guaranteed delivery messaging platform. Intuitively it means that when computer A sends a message to computer B the latter is not required to be operational. Once B goes online the messages will be delivered.

While JMS and in particular, Joram JMS, requires configuration files for running individual JMS hosts, we have decided to use Jade-HTTP protocol to implement communicator platforms. The Jade-HTTP has only minimal configuration requirements allowing for a creation of ad-hoc networks of agents.

B.1.2.2 Main features of Prova-AA

The main design philosophy behind Prova-AA is the minimalism and simplicity of the introduced syntax extensions. Prova-AA provides three main constructs for enabling agent communication: sendMsg predicates, reaction rcvMsg rules, and rcvMsg or rcvMult inline reactions.

- sendMsg predicate

The sendMsg predicate can be embedded into the body of an arbitrary derivation or reaction rule. It can fail only if the parameters are incorrect and the message could not be sent due to various other reasons including the dropped connection (note that in the JMS case, the message may be sent anyway even if the network is down).

The format of the predicate is:

```
    sendMsg(Protocol,Agent,Performative,[Predicate|Args]|Context)
or
    sendMsg(Protocol,Agent,Performative,Predicate(Args)|Context)
```

where Protocol can currently be either self, jade, jms, or queue. It is either the message to self, a message sent to a message queue of a particular agent running locally in another thread (queue), or a message sent via jms or jade communication protocols. Agent denotes the target of the message. For self, jade, and jms methods, Agent is the name of the target agent. For queue message, Agent is the object representing the message queue of the target agent. For jade messages, the agent name takes the form agent@machine while for jms messages the agent locations are read from configuration files and are not specified in the Agent parameter. Performative corresponds to the semantic instruction the broad characterization of the message. A standard nomenclature of performative is FIPA Agents Communication Language ACL.

[Predicate|Args] corresponds to the bracketed form and Predicate(Args) corresponds to functional form of the message content sent in the message envelope. The first form can be useful to match any literal including arity-0 predicates (in which case, query() is the represented as [query]) or arity-1 predicates (in which case, query(arg1) is represented as [query,arg1]). The problem with the functional form is that it is impossible to specify a general pattern accommodating predicates of arbitrary arity while the bracketed version is compatible with any arity. Context includes an arbitrary length list of comma-separated parameters that can be used to identify the message or to distinguish the replies to this particular message from

[1] Java Agent Development framework, JADE, http://sharon.cselt.it/projects/jade].

other messages. In particular, it can be useful to include the protocol as part of context for the recipient of the message to be able to reply by using the same protocol.

The following code shows a complete rule that sends a code base (a fragment of Prova code) from an external File to the agent Remote that will then assimilate the rules being sent. The rules are encapsulated in a serializable Java StringBuffer object and sent with the literal for the built-in predicate consult. The particular version of consult will then read on the Remote machine the Prova statements from a StringBuffer (in contrast to the standard version of consult that reads statements from the specified file provided as an input string).

```
% Upload a rule base read from File to the host at address Remote
%via Prova-JMS

upload_mobile_code(Remote,File) :-
    % Opening a file returns an instance of java.io.BufferedReader in Reader
    fopen(File,Reader),
    Writer = java.io.StringWriter(),
    copy(Reader,Writer),
    Text = Writer.toString(),
    % SB will encapsulate the whole content of File
        SB = StringBuffer(Text),
    sendMsg(jade,Remote,eval,consult(SB)).
```

Before the discussion below about the reaction rules, it should be mentioned that messages for performatives reply, return, and end_of_transmission are assumed by the runtime engine to continue with the encompassing conversation preserving an internally maintained conversation-id. For all other messages, a message is assumed to open a new conversation and is internally assigned a unique conversation-id. The mechanism of conversation-id(s) is designed in such a way that the returned messages can be transparently and uniquely matched against inline reactions patterns (see below).

- Reaction rcvMsg rules

The target agent reacts to the message based on its pattern including the protocol, sender, performative, message content, and context. Reaction rules are special derivation rules for a reserved predicate rcvMsg. The syntax of the rcvMsg parameters is exactly the same as that for the sendMsg parameters except for the Agent parameter corresponding in the rcvMsg case to the agent from which the message has arrived.

The following code shows a general purpose but simplified reaction rules for queryref messages. The first rule triggers a non-deterministic derivation of the literal [X|Xs] sent as the message content by using the local rules and replies to the queryref originator with as many messages with the performative reply as there are possible instantiations to [X|Xs]. The second rule sends a special end_of_transmission message that the originator of the queryref message can use to determine that the sequence of replies is complete. The Protocol parameter available as the first parameter allows the recipient of queryref to know the protocol (jade, jms etc.) that should be used for replies.

```
% Reaction rule to general queryref
rcvMsg(Protocol,From,queryref,[X|Xs]|Context) :-
    derive([X|Xs]),
```

```
    sendMsg(Protocol,From,reply,[X|Xs]|Context).
rcvMsg(Protocol,From,queryref,[X|Xs],Protocol) :-
    sendMsg(Protocol,From,end_of_transmission,[X|Xs]|Context).
```

Example B.3 *(Lost Updates) The following example implements three communicating agents which illustrate the need for concurrency control.It consists of the following of a bank agent and two customer agents.*

- *The bank agent has (Listing B.3) a knowledge base with facts "balance" storing the name of the holder and amount and a method to update the amount of a bank account. It also has a reaction rule, which returns the amount on request or updates it in the event of receiving a corresponding message from an external agent.*
- *The first customer agent (Listing B.4) has two methods "credit" and "debit" with two arguments account holder and amount. The example illustrates that "credit" needs to query the bank agent for the amount, then it should wait for some time and then it should calculate the new amount and send a message to the bank requesting an update of the account.*
- *The second customer agent (Listing B.5) credits on the same account upon start concurrently to illustrate a lost update.*

Listing B.3: Lost updates: The bank agent

```
2  rcvMsg(XID, Protocol, From, inform, update(Type, Name, Amount, NewBalance)) :-
       derive(update(Type, Name, Amount, NewBalance)),
4      sendMsg(XID, Protocol, From, reply, update(Type, Name, Amount, NewBalance)).

6  balance("adila", 100).
   balance("wikan", 200).
8  lock(false).

10
   update(Name, NewBalance) :-
12     retractall(balance(Name, _)),
       assert(balance(Name, NewBalance)).
14
   rcvMsg(XID, Protocol, From, inform, update(Name, NewB)) :-
16     derive(update(Name,NewB)),
       sendMsg(XID, Protocol, From, reply, update(Name, NewB)).
18
   rcvMsg(XID, Protocol, From, inform, lock()) :-
20     retractall(lock(_)),
       assert(lock(true)),
22     sendMsg(XID, Protocol, From, reply, lock(true)).

24 rcvMsg(XID, Protocol, From, inform, unlock()) :-
       retractall(lock(_)),
26     assert(lock(false)),
       sendMsg(XID, Protocol, From, reply, lock(false)).
28

30 rcvMsg(XID, Protocol, From, queryref, [X|Xs]) :-
       println(["queryref received.", X | Xs]),
32     derive([X|Xs]),
       println(["queryref derived.", X | Xs]),
34     sendMsg(XID, Protocol, From, reply, [X|Xs]).
   rcvMsg(XID, Protocol, From, queryref, [X|Xs]) :-
36     sendMsg(XID, Protocol, From, end_of_transmission, [X|Xs]).
```

Listing B.4: Lost updates: Client 1

```
1
```

```
:-eval(do_transaction()).

% The format is agent@machine where the machine is the target machine we are communicating ...
    ...with
remote("server@bioinf3").

rcvMsg(XID,Protocol,From,end_of_transmission,[X|Xs]|LocalContext).

do_transaction() :-
    transfer("adila", "wikan", 30).

credit(Name, Amount) :-
    remote(Remote),
    sendMsg(XID, jade, Remote, queryref, balance(Name, Balance)),
    rcvMsg(XID, jade, Remote, reply, balance(Name, Balance)),
    println(["Balance before credit: ", Name, "=", Balance]),

    java.lang.Thread.sleep(2000),

    NewBalance = Balance + Amount,
    sendMsg(XID2, jade, Remote, inform, update(Name, NewBalance)),
    rcvMsg(XID2, jade, Remote, reply, update(Name, _)),

    java.lang.Thread.sleep(1000),

    sendMsg(XID3, jade, Remote, queryref, balance(Name, B)),
    rcvMsg(XID3, jade, Remote, reply, balance(Name, B)),
    println(["Balance after credit: ", Name, "=", B]).


debit(Name, Amount) :-

    acquire_lock(),

    remote(Remote),
    sendMsg(XID, jade, Remote, queryref, balance(Name, Balance)),
    rcvMsg(XID, jade, Remote, reply, balance(Name, Balance)),
    println(["Balance before debit: ", Name, "=", Balance]),

    java.lang.Thread.sleep(2000),

    NewBalance = Balance - Amount,
    sendMsg(XID2, jade, Remote, inform, update(Name, NewBalance)),
    rcvMsg(XID2, jade, Remote, reply, update(Name, _)),

    java.lang.Thread.sleep(1000),

    sendMsg(XID3, jade, Remote, queryref, balance(Name, B)),
    rcvMsg(XID3, jade, Remote, reply, balance(Name, B)),
    println(["Balance after debit: ", Name, "=", B]),

    release_lock().

transfer(From, To, Amount) :-

    acquire_lock(),

    remote(Remote),
    sendMsg(XID, jade, Remote, queryref, balance(From, _)),
    rcvMsg(XID, jade, Remote, reply, balance(From, B1)),

    NewB1 = B1 - Amount,
    sendMsg(XID2, jade, Remote, inform, update(From, NewB1)),
    rcvMsg(XID2, jade, Remote, reply, update(From, NB1)),

    java.lang.Thread.sleep(2000),

    sendMsg(XID3, jade, Remote, queryref, balance(To, _)),
    rcvMsg(XID3, jade, Remote, reply, balance(To, B2)),
```

```
71      NewB2 = B2 + Amount,
        sendMsg(XID4, jade, Remote, inform, update(To, NewB2)),
73      rcvMsg(XID4, jade, Remote, reply, update(To, NB2)),
        println(["Before transfer:  ", From, "=", B1, "  ", To, "=", B2]),
75      println(["After  transfer:  ", From, "=", NB1, "  ", To, "=", NB2]),

77      release_lock().

79  acquire_lock() :-
        remote(Remote),
81      sendMsg(XID, jade, Remote, queryref, lock(_)),
        rcvMsg(XID, jade, Remote, reply, lock(false)),
83  %   Status = true,
        println(["lock acquired"]),
85      sendMsg(XID2, jade, Remote, inform, lock()),
        rcvMsg(XID2, jade, Remote, reply, lock(true)),
87      println(["account locked"]),
        !.
89  acquire_lock() :-
        remote(Remote),
91      sendMsg(XID, jade, Remote, queryref, lock(_)),
        rcvMsg(XID, jade, Remote, reply, lock(true)),
93  %   Status != true,
        println(["lock not acquired, sleeping"]),
95      java.lang.Thread.sleep(1000),
        acquire_lock().
97
    release_lock() :-
99      remote(Remote),
        sendMsg(XID2, jade, Remote, inform, unlock()),
101     rcvMsg(XID2, jade, Remote, reply, lock(false)),
        println(["account released"]).
```

Listing B.5: Lost updates: Client 2

```
2   :-eval(do_transaction()).

4   % The format is agent@machine where the machine is the target machine we are communicating ...
        ...with
    remote("server@bioinf3").
6

8   % A testing harness for printing incoming end_of_transmission messages.
    % rcvMsg(XID,Protocol,From,end_of_transmission,[X|Xs]|LocalContext) :-
10  %   println(["end_of_transmission for conversation-id ",XID,": ",[X|Xs]," "|LocalContext]).
    rcvMsg(XID,Protocol,From,end_of_transmission,[X|Xs]|LocalContext).
12

14  do_transaction() :-
    %    debit("adila", 20),
16      report_all().

18  report_all() :-
        acquire_lock(),
20      report_account(["wikan", "adila"], 0),
        release_lock().
22
    credit(Name, Amount) :-
24      remote(Remote),

26      sendMsg(XID, jade, Remote, queryref, balance(Name, Balance)),
        rcvMsg(XID, jade, Remote, reply, balance(Name, Balance)),
28      println(["Balance before credit: ", Name, "=", Balance]),

30      java.lang.Thread.sleep(2000),
    %   println(["Line after sleep"]),
32
        NewBalance = Balance + Amount,
```

```
34      sendMsg(XID2, jade, Remote, inform, update(Name, NewBalance)),
        rcvMsg(XID2, jade, Remote, reply, update(Name, _)),
36 %    println(["Old: ",Balance, "   New: ", NewB]),

38      java.lang.Thread.sleep(1000),
   %    println(["Line after sleep2"]),
40
        sendMsg(XID3, jade, Remote, queryref, balance(Name, B)),
42      rcvMsg(XID3, jade, Remote, reply, balance(Name, B)),
        println(["Balance after credit: ", Name, "=", B]).
44

46 debit(Name, Amount) :-

48      acquire_lock(),

50      remote(Remote),
        sendMsg(XID, jade, Remote, queryref, balance(Name, Balance)),
52      rcvMsg(XID, jade, Remote, reply, balance(Name, Balance)),
        println(["Balance before debit: ", Name, "=", Balance]),
54
        java.lang.Thread.sleep(1000),
56 %    println(["Line after sleep"]),

58      NewBalance = Balance - Amount,
        sendMsg(XID2, jade, Remote, inform, update(Name, NewBalance)),
60      rcvMsg(XID2, jade, Remote, reply, update(Name, _)),
   %    println(["Old: ",Balance, "   New: ", NewB]),
62
        java.lang.Thread.sleep(1000),
64 %    println(["Line after sleep2"]),

66      sendMsg(XID3, jade, Remote, queryref, balance(Name, B)),
        rcvMsg(XID3, jade, Remote, reply, balance(Name, B)),
68      println(["Balance after debit: ", Name, "=", B]),

70      release_lock().

72
   report_account([], Sum) :-
74      println(["Sum of all account=", Sum]).
   report_account([N|Ns], Sum) :-
76      remote(Remote),
        sendMsg(XID, jade, Remote, queryref, balance(N, _)),
78      rcvMsg(XID, jade, Remote, reply, balance(N, B)),
        NSum = Sum + B,
80      println([N, "=", B, "   ", NSum]),
        report_account(Ns, NSum).
82
   acquire_lock() :-
84      remote(Remote),
        sendMsg(XID, jade, Remote, queryref, lock(_)),
86      rcvMsg(XID, jade, Remote, reply, lock(false)),
   %    Status = true,
88      println(["lock acquired"]),
        sendMsg(XID2, jade, Remote, inform, lock()),
90      rcvMsg(XID2, jade, Remote, reply, lock(true)),
        println(["account locked"]),
92      !.
   acquire_lock() :-
94      remote(Remote),
        sendMsg(XID, jade, Remote, queryref, lock(_)),
96      rcvMsg(XID, jade, Remote, reply, lock(true)),
   %    Status != true,
98      println(["lock not acquired, sleeping"]),
        java.lang.Thread.sleep(1000),
100     acquire_lock().

102 release_lock() :-
        remote(Remote),
```

```
104     sendMsg(XID2, jade, Remote, inform, unlock()),
        rcvMsg(XID2, jade, Remote, reply, lock(false)),
106     println(["account released"]).

108 Results:
    ---bank
110 queryref received.lock<java.lang.Object._-834369677-978847522>
    queryref derived.lockfalse
112 queryref received.lock<java.lang.Object._-6905464631609046756>
    queryref derived.lockfalse
114 queryref received.balanceadila<java.lang.Object.N@@9>
    queryref derived.balanceadila100
116 queryref received.balancewikan<java.lang.Object.N@@13>
    queryref derived.balancewikan200
118 queryref received.lock<java.lang.Object._517604657-1267444982>
    queryref derived.lockfalse
120 queryref received.lock<java.lang.Object._1331913378-1153084111>
    queryref derived.lockfalse
122 queryref received.balancewikan<java.lang.Object.N@@17>
    queryref derived.balancewikan230
124 queryref received.balanceadila<java.lang.Object.N@@19>
    queryref derived.balanceadila70
126
    ---client 1
128 lock acquired
    account locked
130 Before transfer:  adila=100  wikan=200
    After  transfer:  adila=70  wikan=230
132 account released
    ----cleint2
134 lock acquired
    account locked
136 wikan=230   230
    adila=70   300
138 Sum of all account=300
    account released
```

- Inline reaction rules

If one considers the problem of an agent communicating asynchronously with several agents at the same time, one must appreciate the difficulty in recognizing the incoming messages that can look exactly the same but belong to different communication protocols or different stages of the same communication protocol. It is also important when specifying the code in the form of rules to maintain the context (or state) of the particular conversation.

Inline reactions simplify the programming style for specifying communication protocols by offering the user the opportunity to insert the literals for the predicate rcvMsg directly into the body of the rules. The syntax of the inline rcvMsg parameters is the same as when rcvMsg occurs in the head of the reaction rules.

The following example contrasts two possible ways of invoking a query for the predicate gopubmed locally and remotely. In the case of a local query, the gopubmed predicate is invoked directly from the body of the rule. This results in the instantiation of the result variable SB, which is communicated back to the invoking object Obj passed as the first parameter in the head of the rule. In the case of a remote query, a pair of sendMsg/rcvMsg predicates replaces this local call with the asynchronous remote call to the target agent prova@rocket.

```
manager("prova@rocket").

local_query_gopubmed(Obj,X,MaxResults,Flags) :-
    gopubmed(X,MaxResults,Flags,SB),
```

```
    Obj.results_ready(SB).

remote_query_gopubmed(Obj,X,MaxResults,Flags) :-
    manager(Manager),
    sendMsg(jade,Manager,queryref,gopubmed(X,MaxResults,Flags,SB),id0),
    rcvMsg(jade,Manager,reply,gopubmed(X,MaxResults,Flags,SB),id0),
    Obj.results_ready(SB).
```

It should be noted that the inference engine does not wait on the rcvMsg call. Instead, the execution flow is stored in a transparently constructed temporal reaction rule that is activated once the message matching the specified pattern in rcvMsg has arrived. The engine then continues processing other incoming messages and goals. The temporal rule includes the pattern of the message specified in rcvMsg (or rcvMult, see below) in the head of the rule and all remaining goals at the current execution point in its body. This behaviour is different from suspending the current thread and waiting for the replies to arrive as implemented by some programming languages such as Go!. The Prova-AA execution engine maintains one main thread while allowing an unlimited number of conversations to proceed at the same time without incurring the penalty and limitations of multiple threads otherwise used solely for the purpose of maintaining several conversation states at the same time.

The rcvMsg inline reaction should normally be used only in situation when zero or one replies are expected. According to the semantics shown for the (outline) queryref reaction rules above, a special end_of_transmission message serves as an indicator that no more replies will be arriving. If the queryref returns no replies because the queried predicate fails, end_of_transmission will still be sent to the originator of the query. In the case of a waiting rcvMsg inline reaction in the form of its corresponding temporal rule, the end_of_transmission results in discharging of the temporal rule and failure of the literals that follow rcvMsg. For a single result, the temporal rule corresponding to rcvMsg is deleted upon the receipt of the reply, and end_of_transmission is simply ignored.

If more than one reply is sent back to the query originator rcvMsg will be discharged after the first reply and all subsequent replies and end_of_transmission will be ignored unless there are (outline) reaction rules that will match their pattern. The last Prova-AA construct rcvMult is introduced in order to deal with the multiple replies situation. The arguments of rcvMult are exactly the same as for rcvMsg but the semantics is different. The temporal rule corresponding to rcvMult continues to wait for incoming messages until the end_of_transmission arrives. For each incoming reply, the trail of outstanding goals in the search tree will be independently explored until exhaustion or until a new rcvMsg or rcvMult reaction is encountered.

The following fragment shows how a remote query to predicate parent possibly returning multiple replies results in non-deterministic exhaustive printing of all the replies until finally, end_of_transmission arrives and rcvMult is discharged.

```
test() :-
% Send a queryref message with replies to be sent back to the agent
sendMsg(jade,Me,queryref,parent(X,Y),id0),
rcvMult(jade,Me,reply,parent(X,Y),id0), println(["Inline reaction
",rcvMult(Me,reply,parent(X,Y),id0)]).
```

Returning to the issue of transparent processing of conversation-id(s), it should be noted that in the above example, rcvMult will match only the reply to the sendMsg immediately preceding it.

If there was another non-reply message sent between these two messages (thereby initiating another conversation) the rcvMult would be implicitly assigned to this messages conversation and would not have matched the reply. In summary, the current implementation treats send/reply pairs as belonging to the same conversation and does not allow any other conversation initiating messages to occur between them. In the future, this design may be changed to include explicit treatment of conversation-id(s).

B.1.3 Communicator Prova-AA agent for Java applications

We have developed and included with the Prova-AA distribution a special class reagent Communicator that allows arbitrary Java applications to embed a Prova-AA agent and become equal participants in the network of such agents. In this section we describe an example of a Communicator Prova-AA agent.

The following example shows a fragment of a class BioAgent that instantiates an embedded version Prova-AA Communicator that can exchange messages via JADE-HTTP on port 7779. The Communicator is constructed by calling its constructor with two arguments: the port number and the file with Prova script with initial goals, facts and rules constituting its initial rule base. The public variable queueAgent inside a communication represents a message queue for the Prova engine. The queue insertions and deletions are synchronised by the class RMessageQueue inside Prova.

```
final private static String rules = "requestor.prova";
protected Communicator comm;
protected RMessageQueue queueAgent;
public void results_ready( StringBuffer sb ) {
         System.out.println( sb );
}
public void submit_gopubmed( String s, int imaxpapers, int imode ) {
  RMessage r = new RMessage("");
  Integer maxpapers = new Integer(imaxpapers);
  Integer mode = new Integer(imode);
  r.append_string("[0,\"eval\",[query_gopubmed,")
      .append(this)
      .append_string(",")
      .append(s)
      .append_string(",")
      .append(maxpapers)
      .append_string(",")
      .append(mode)
      .append_string("]]");
  queueAgent.add( r );
}
public static void main(String[] args) {
  // ...
  comm = new Communicator( "7778", rules );
  queueAgent = comm.queueAgent;
}
```

The Java application interfaces with the Prova rule base by using this message queue for submitting goals and using methods similar to the method results_ready shown above to receive results of various asynchronous queries inside the Java or Web application. The distributed GoPubMed web application embeds a Communicator agent in this way. The listing below shows how easy a java application can programmatically issue queries to the running GoPubMed system. The listing is the full text of the requestor.prova file used for initialising the Communicator agent in the BioAgent class as shown above.

```
gopubmed_server("prova@server_name").

query_gopubmed(BioAgent,X,MaxResults,Flags) :-
gopubmed_server(GoPubMed),
sendMsg(XID,jade,GoPubMed,queryref,gopubmed(X,MaxResults,Flags,SB)),
rcvMsg(XID,jade,GoPubMed,inform,gopubmed(X,MaxResults,Flags,SB)),
BioAgent.results_ready(SB).
```

The fact gopubmed_server stores the agent address of the Prova-AA agent embedded in the GoPubMed web application. The query for the predicate query_gopubmed is called from the Java application represented by the class BioAgent. In the body of the corresponding rule the server address is retrieved and a pair sendMsg/rcvMsg is used to query the predicate gopubmed inside the GoPubMed server agent. Upon the receipt of the reply the method results_ready in BioAgent is called returning a StringBuffer SB with the HTML code for the produced page.

B.1.3.1 Specifying the behaviour of Prova-AA agents as state machines

In this section, we consider a more rigorous, state machines based implementation of the roles the Prova-AA agents play in conversations. Consider two agent roles Seller and Buyer engaging in a Direct Buy protocol. The following Prova-AA listings show the specifications and implementations of the Seller and Buyer roles. The code is included with the Prova distribution as test049.prova. The general pattern for encoding states and transitions in Prova-AA is:

```
state_j(...conversation_paramaters...) :-
      [!,],
      [event_j+1(...),] OR [condition_j_j+1(...), !,]
      [actions_j+1(...),]
      [state_j+1(...)].
state_j(...) :-
      ...
state_j+1(...) :-
     ...
```

For each state j in the state machine of an agent role, possibly parameterised with additional conversation parameters passed from the preceding states, we specify a set of rules for predicate state_j. If there is a family of states that are chosen dynamically based on their parameters, we specify one rule for state_j with the particular parameters instantiation. Moreover, if there are N transitions from state_j to subsequent states, there is one rule for state_j per transition. Each rule for state_j contains in its body:

- possible Cut (!) used when parameters of state_j select this particular parameterised state;
- EITHER the event for a transition j ⇒j+1 using rcvMsg or rcvMult inline reactions;

- OR the conditions for this transition;
- the actions that should be executed on entry to state j+1,
- and the call to next state j+1.

If both event and condition are required and there could be multiple different events leading to different states, we would normally introduce an intermediate state before taking a further transition guarded by the condition. If there is no multiple choice involved in the event, event and condition(s) can be combined in one rule. Consider for example, the two rules for the directbuy_seller_1 corresponding to state 1 in Seller. There are two rules because there are two transitions from state 1. The first rule checks if the product with name Product exists in the database. If the product is there, as is the case for car(volvo,s60), the Cut follows and the two messages corresponding to the entry actions for state directbuy_seller_2(yes) are executed. Otherwise, a transition to directbuy_seller_2(no) with the corresponding message sending follows. Consider the transitions from state 1 in Buyer to states 2(agree) and 2(refuse). The following line rcvMsg(XID,Protocol,From,Perf,Product,seller) allows the agent to wait for a message for an unspecified performative Perf. The family of states 2 is parameterised by the particular incoming message (agree or refuse) that are chosen by unification of the predicate parameters in the heads of the corresponding rules.

```
%Seller:
:- eval(directbuy_seller_0()).
product(car(volvo,s60),11000). product(car(ford,fiesta),5000).

directbuy_seller_0() :-
 rcvMult(XID,Protocol,From,query_ref,Product,buyer),
 directbuy_seller_1(XID,Protocol,From,Product).

directbuy_seller_1(XID,Protocol,From,Product) :-
 product(Product|_), !,
 sendMsg(XID,Protocol,From,agree,Product,seller),
 sendMsg(XID,Protocol,From,inform_result,Product,seller),
 directbuy_seller_2(yes,XID,Protocol,From,Product).

directbuy_seller_1(XID,Protocol,From,Product) :-
 sendMsg(XID,Protocol,From,refuse,Product,seller),
 directbuy_seller_2(no,XID,Protocol,From,Product).

directbuy_seller_2(yes,XID,Protocol,From,Product) :- !,
 rcvMsg(XID,Protocol,From,request,[Product,Price],buyer),
 product(Product,SellerPrice),
 directbuy_seller_3(XID,Protocol,From,Product,Price,SellerPrice).

directbuy_seller_2(no,XID,Protocol,From,Product).

directbuy_seller_3(XID,Protocol,From,Product,Price,SellerPrice) :-
 Price>SellerPrice,!,
 sendMsg(XID,Protocol,From,agree,[Product,Price],seller),
 sendMsg(XID,Protocol,From,inform_done,[Product,Price],seller),
```

```
 directbuy_seller_4(agree,XID,Protocol,From,Product,Price).

directbuy_seller_3(XID,Protocol,From,Product,Price,SellerPrice) :-
 sendMsg(XID,Protocol,From,refuse,[Product,Price],seller),
 directbuy_seller_4(refuse,XID,Protocol,From,Product,Price).

directbuy_seller_4(agree,XID,Protocol,From,Product,Price).
directbuy_seller_4(refuse,XID,Protocol,From,Product,Price).
```

B.2 Organising Travels Scenario

In this section, for illustration purposes, we show a PROVA implementation of the travel Scenario described in Deliverable I5-D2 [3]. The program consists of two agents a Travel Agency agent and a customer agent. The implementation follows the main system requirements described in the Scenario in section 5.2.1 of the deliverable. In order for the system to accomplish initial planning, the following are required

- Web query language with advanced reasoning capabilities
- support for receiving volatile data
- query and reasoning with volatile data

Further details on the specification of the scenario and use cases are to be found at [3].

B.2.1 Initial Planning

B.2.1.1 Gathering and reasoning with information

Listing 7 contains the Travel Agency agent where information about flights is implemented as facts, it contains information about origin and destination cities, dates and times of departure and arrival, class and prices in (lines 1-14). Gathering information and querying the content of the web can also be realized using PROVA as shown in listing 6. The program issues a query to the web which then retrieves the XML documents of the query output (Hotels in Rome in this example) and then parse the XML documents for needed information such as name of the hotel and availability.

Listing B.6: Organising Travels Scenario: Travel Agency:with web connection

```
1  :-eval(doSearch()).

3  searchURL("http:///travel/airlines/lang/en-us/itinerary.asp?").

5
   doSearch():-
7      searchDestination("Rome",10,XML),
       printXML(XML).
9

11 searchDestination(Query,Restrict,XML):-
       print(["Query for ",Query," at hotels.net"]),
13     searchURL(BaseURL),
       concat([BaseURL,"..."],URLString0),
15     concat([URLString0,"..."],URLString1),
       ...
17     retrieveXML(URLString,XML),
       println(["URL Search:", URLString]),
```

```
19      println(["done"]).

21  %%%%%%%%%%%%%%%% UTILITIY predicates %%%%%%%%%%%%%%%%%%%%%%

23  retrieveXML(URLString,Root):-
        URL = java.net.URL(URLString),
25      print(["."]),
        Stream = URL.openStream(),
27      print(["."]),
        ISR = java.io.InputStreamReader(Stream),
29      XMLResult = XML(ISR),
        Root = XMLResult.getDocumentElement(),
31      print(["."]).

33  printXML(XML):-
        println(["---------------------------------------------------"]),
35      descendants(XML, "Accomodation", City),
        descendantValue(City, "City", City),
37      descendantValue(Name, "Hotel_Name", Name),

39      println(["CITY:", CITY]),
        println(["NAME:", Name]),
41
        println(["---------------------------------------------------"]).
43
    handle_exception(Text) :-
45      exception(Ex),
        println([Ex]),
47      Text = 'compensation'.

49  descendantValue(Node,Name,StringName):-
        descendantNode(Node,Name,Data),
51      StringName = Data.getNodeValue().

53  descendantNode(Node,Name,Data):-
        X = Node.getElementsByTagName(Name),
55      X.nodes(ID),
        Data = ID.getFirstChild().
57


59
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
61
    % Definition of a descendant in any depth
63  % XPath equivalent: //*
    descendants(Node,Node).
65  descendants(Element,S2):-
        children(Element,S1),
67      descendants(S1,S2).

69  % Definition of a descendant in any depth with given name
    % XPath equivalent: //Name
71  descendants(Node,Name,Descendant):-
        descendants(Node,Descendant),
73      nodeName(Descendant,Name).

75  descendantsValueContains(Current,Name,ValuePart):-
        descendants(Current,Name,Node),
77      nodeValue(Node,Value),
        Value.contains(ValuePart).
79
    % Simulates an XPath traversal
81  % XPath equivalent: //Name/text()
    descendantsValue(Current,Name,Value):-
83      descendants(Current,Name,Node),
        nodeValue(Node,Value).
85
    descendantsIntValue(Current,Name,Value):-
87      descendants(Current,Name,Node),
        nodeValue(Node,StringValue),
```

```
89      Value = Integer.parseInt(StringValue).

91  children(Element,Child):-
        Childs = Element.getChildNodes(), % Java call returning direct children
93      Childs.nodes(Child).

95  % Definition of a child with a given name
    % XPath equivalent: /Name
97  children(Node,Name,Child):-
        children(Node,Child),
99      nodeName(Child,Name).

101
    printNode(Node):-
103     nodeName(Node,Name),
        nodeValue(Node,Value),
105     println([Name," = {",Value,"}"]).

107 nodeName(Node,Name):-
        Name = Node.getNodeName().
109
    nodeValue(Node,Value):-
111     Data = Node.getFirstChild(),
        Raw = Data.getNodeValue(),
113     Value = Raw.trim().
```

B.2.1.2 Arranging the trip according to plan

Systems that could arrange a trip on the web according to a given plan require:

- Communication of data between systems that cooperate to arrange a trip
- Updates to persistent data

The Travel Agency agent also manages the communication between the two agents by exchanging notifications and invoking reaction rules to produce alternatives plans if initial plan doesn't hold any more because of a change in the plan conditions such as flight delay as shown in the example below (lines 48-50).

Listing B.7: Organising Travels Scenario: Travel Agency

```
    % == flights ============================================================
2   %       No.     From       To         F.Time    To Time   Date         Class     Price
    flight(fl001, "Munich", "Milian", "5-3-2005/09:30", "5-3-2005/12:00",
4       "economy", 70).
    flight(fl002, "Milian", "Florence", "12-3-2005/14:45", "12-3-2005/17:00",
6       "economy",30).
    flight(fl003, "Munich", "Rome", "5-3-2005/10:00", "5-3-2005/12:00",
8       "economy", 60).
    flight(fl004, "Florence", "Munich", "15-3-2005/09:30", "15-3-2005/12:00",
10      "economy",65).
    flight(fl005, "Milian", "Florence", "12-3-2005/17:42", "12-3-2005/19:40",
12      "economy",35).
    flight(fl006, "Florence", "Munich", "15-3-2005/10:30", "15-3-2005/13:00",
14      "economy",95).

16  % requests for all flights
    rcvMsg(ID, Protocol, From, query, [flights|Rest]) :-
18      println(["got request for flights"]),
        buildFlightList(),
20      flight_list(L),
        sendMsg(ID, jade, From, inform, flights(L)).
22
    % builds a list of flights that the agent sends afterwards to the client
24  buildFlightList() :-
        retractall(flight_list(_)),
```

```
26     assert(flight_list([])),
       flight(ID, From, To, FromDate, ToDate, Class, Costs),
28     flight_list(List),
       retractall(flight_list(_)),
30     assert(flight_list([[ID, From, To, FromDate, ToDate, Class, Costs]|List])),
       fail().
32  buildFlightList().

34  % == simulation ============================================================

36  % change of the time within tolerable limits
    rcvMsg(ID, Protocol, From, query, [simulate_flight_change_1|Rest]) :-
38     debug("simulating plan change ..."),
       sendMsg(ID, jade, From, inform, flight_change([fl002, "Milian", "Florence",
40         "12-3-2005/14:55", "12-3-2005/17:20", "economy",30])).

42  % change of the day which is not tolerable
    rcvMsg(ID, Protocol, From, query, [simulate_flight_change_2|Rest]) :-
44     debug("simulating plan change ..."),
       retractall(flight(fl002,_,_,_,_,_,_)),
46     assert(flight(fl002, "Milian", "Florence", "16-3-2005/15:23",
           "16-3-2005/18:00", "economy",46)),
48     sendMsg(ID, jade, From, inform, flight_change([fl002, "Milian", "Florence",
           "16-3-2005/15:23", "16-3-2005/18:00", "economy",46])).
50
    % cancellation of a flight
52  rcvMsg(ID, Protocol, From, query, [simulate_flight_cancel|Rest]) :-
       debug("simulating plan cancellation"),
54     sendMsg(ID, jade, From, inform, flight_canceled(fl004)).

56  % == auxiliary rules =======================================================

58  % two rules for the debugging output. the 1st takes lists of strings and the
    % 2nd works with simple strings
60  % params:   in:LIST     list of strings you want to print out
    debug([S|R]) :-
62     !,
       concat(R,T),
64     concat([S,T],F),
       debug(F).
66  % params:   in:String   the text you want to print out
    debug(S) :-
68     println(["[DEBUG] ",S]).
```

B.2.2 Arranging the trip

Listing 8 shows the user agent, first the user trip preferences is implemented along with the hotels and trains sample knowledge bases (a sample for the example purposes). It also contains the reactions rules that produced alternative plans that still fits the users time frame and preferences.

B.2.3 Adapt Plan to changes

The Travel agency agent notify the customer agent about changes in the flight schedule, the customer agent reacts by searching for the best alternative plans like shown in the listing below.

Listing B.8: Organising Travels Scenario: customer

```
   :-eval(test()).
2
   % == general settings ======================================================
4  flight_agent("flight@bioinf3"). % agent name of flight agency
```

```
 6 % == trains ==============================================================
   %     No.    From     To        F.Time   To Time  Date        Class      Price
 8 train(tr001, "Rome", "Venice", "8-3-2005/11:11", "8-3-2005/13:12",
       "economy", 10).
10 train(tr002, "Venice", "Milian", "9-3-2005/13:21", "9-3-2005/16:16",
       "economy", 12).
12 train(tr003, "Rome", "Venice", "11-3-2005/11:11", "11-3-2005/13:23",
       "economy", 15).
14
   % == hotels ==============================================================
16 %     No.     Where   Name Stars 2RF  1RF  Price p.r.
   hotel(htl001, "Rome", "Rome Plaza", "4", "1", "1", "60").
18 hotel(htl002, "Rome", "Rome In", "3", "1", "1", "20").
   hotel(htl003, "Venice", "Hilton Venice", "5", "1", "1", "70").
20 hotel(htl004, "Milian", "Milian-In", "3", "1", "1", "20").
   hotel(htl005, "Florence", "Continental Florence", "4", "1", "1", "30").
22
   % == preferences =========================================================
24 tripMinStars(3).
   tripFrom("Munich").
26 tripStart("5-3-2005/9:00").
   tripEnd("15-3-2005/17:00").
28 tripStations(["Venice", "Florence", "Rome", "Milian"]).
   tripMaxCosts(400).
30
   % == reaction rules ======================================================
32
   % reacts to changing in flights
34 rcvMsg(ID, jade, From, inform, [flight_change, Data|_]) :-
       println([""]),
36     Data = [FID, FFrom, To, FromDate, ToDate, Class, Costs],
       debug(["something in flight ", FID, " has changed ... "]),
38     debug(["  flight No. ", FID]),
       debugl("     was: "),
40     print_flight(FID),
       retractall(flight(FID,_,_,_,_,_,_)),
42     assert(flight(FID, FFrom, To, FromDate, ToDate, Class, Costs)),
       debugl("     is:  "),
44     print_flight(FID),
       validate_replan().
46
   % reacts to cancellation of flights
48 rcvMsg(ID, jade, From, inform, [flight_canceled, FID|_]) :-
       println([""]),
50     debug(["flight No.", FID, " was cancelled, ..."]),
       retractall(flight(FID,_,_,_,_,_,_)),
52     retractall(current_plan(_)),
       validate_replan().
54
   % == planning related stuff (core) =======================================
56
   % finds the best (cheapest) plan of all
58 best_plan(Plan, Sum) :-
       retractall(current_best_plan(_,_)),
60     assert(current_best_plan([], 99999)),
       plan(CPlan, CSum),
62     check_plan(CPlan),
       current_best_plan(CPlan2, CSum2),
64     egt(CSum2, CSum),
       retractall(current_best_plan(_,_)),
66     assert(current_best_plan(CPlan, CSum)),
       fail().
68 best_plan(Plan, Sum) :-
       current_best_plan(Plan, Sum),
70     lt(Sum, 99999).

72 % builds some simple plan
   % syntax: [[startDate, ID, endDate], ...]
74 plan(Plan, Sum) :-
       % get user's preferences
```

```
76      tripFrom(From),
        tripStart(FromDate),
78      tripEnd(UntilDate),
        tripStations(Cities),
80      tripMaxCosts(PriceLimit),
        % flight from home city into country
82      elementOfList(ToCity, Cities, RestCities),
        % search window of 1 day
84      addDaysToDate(FromDate, 1, NextDate),
        fetch_flight(From, NextDate, ToCity, Costs1, ArrivalDate1, FlightNo1),
86      build_plan_node_from_id(FlightNo1, Node),

88      % flight within country & book some hotels
        fetchRestCities(ToCity,RestCities,ArrivalDate1,PlanRest,SumRest),
90      lastFlight(_, LastDate),
        Plan = [Node|PlanRest],
92      add(SumRest, Costs1, Sum),
        % final calculations and tests
94      dateBeforeDate(LastDate, UntilDate),
        print([""]).
96
    % builds a plan that flies from CurrentCity over all Cities with the costs Sum
98  fetchRestCities(_, [], _, [HotelNode, FlightNode], Costs) :-
        lastCity(LastCity, FromDate),
100     tripEnd(ToDate),
        tripFrom(From),
102     % search for a flight back
        fetch_flight(LastCity, ToDate, From, LastCosts, LastDate, LastNo),
104     build_plan_node_from_id(LastNo, FlightNode),
        cityFromId(LastNo, _, RealLastDate),
106     retractall(lastFlight(_,_)),
        assert(lastFlight(LastNo, RealLastDate)),
108     % search for a hotel for the last days
        fetchHotel(LastCity, FromDate, RealLastDate, HotelId, HotelCosts),
110     build_plan_node_from_id(HotelId, HotelNode),
        add(HotelCosts, LastCosts, Costs).
112 fetchRestCities(CurrentCity, Cities, StartDate, Plan, Sum) :-
        % take some of the cities
114     elementOfList(NextCity, Cities, RestCities),
        % define the search window
116     addDaysToDate(StartDate, 5, NextDate),
        % try to find some train or plane moving from CurrentCity to NextCity
118     fetch_something(CurrentCity,NextDate,NextCity,NextCosts,NextDate1,NextNo),
        build_plan_node_from_id(NextNo, FlightNode),
120     % try to book a hotel from StartDate to NextDate1
        cityFromId(NextNo, _, ToDate),
122     retractall(lastCity(_,_)),
        assert(lastCity(NextCity, ToDate)),
124     fetchHotel(CurrentCity, StartDate, ToDate, HotelId, HotelCosts),
        build_plan_node_from_id(HotelId, HotelNode),
126     fetchRestCities(NextCity,RestCities,NextDate1,RestPlan, RestSum),
        Plan = [HotelNode, FlightNode|RestPlan],
128     add(NextCosts, RestSum, SumZ),
        add(SumZ, HotelCosts, Sum).
130
    % finds something (flight|train) that departs before GFromDate
132 fetch_something(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo) :-
        fetch_train(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo).
134 fetch_something(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo) :-
        fetch_flight(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo).
136
    % finds a flight that departs before GFromDate
138 fetch_flight(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo) :-
        flight(GNo, GFrom, GToCity, GToDate, FromDate, _, GCosts),
140     dateBeforeDate(FromDate, GFromDate).
    % finds a train that departs before GFromDate
142 fetch_train(GFrom, GFromDate, GToCity, GCosts, GToDate, GNo) :-
        train(GNo, GFrom, GToCity, GToDate, FromDate, _, GCosts),
144     dateBeforeDate(FromDate, GFromDate).
    % finds a hotel for the given period
```

```
146 fetchHotel(City, FromDate, ToDate, HotelId, Costs) :-
        hotel(HotelId, City, _, Stars, _, _, Costs),
148     tripMinStars(MinStars),
        toInt(MinStars, IMinStars),
150     toInt(Stars, IStars),
        egt(IStars, IMinStars).
152
    % builds a plan node for the item with the given id
154 build_plan_node_from_id(ID, Node) :-
        hotel(ID, _, _, _, _, _, _),
156     !,
        Node = ["-", ID, "-"].
158 build_plan_node_from_id(ID, Node) :-
        flight(ID, _, _, FDate, TDate, _, _),
160     !,
        Node = [FDate, ID, TDate].
162 build_plan_node_from_id(ID, Node) :-
        train(ID, _, _, FDate, TDate, _, _),
164     !,
        Node = [FDate, ID, TDate].
166

168 % == planning related stuff (validation) ====================================

170 % updates plan's timestamps
    refresh_plan([],[]).
172 refresh_plan([[_,ID,_]|T], [[NStartDate,ID,NEndDate]|NT]) :-
        refresh_step(ID, NStartDate, NEndDate),
174     refresh_plan(T,NT).

176 % returns the current timestamps of item with GNo
    refresh_step(GNo, GFromDate, GToDate) :-
178     flight(GNo, GFrom, GToCity, GToDate, GFromDate, _, GCosts).
    refresh_step(GNo, GFromDate, GToDate) :-
180     train(GNo, GFrom, GToCity, GToDate, GFromDate, _, GCosts).
    refresh_step(HotelId, FromDate, ToDate) :-
182     hotel(HotelId, City, _, Stars, _, _, Costs),
        FromDate = "-",
184     ToDate = "-".

186 % checks whether the plan is valid or not
    check_plan(Plan) :-
188     refresh_plan(Plan, [[StartDate, ID, EndDate]|T]),
        retractall(cp_last_date(_)),
190     assert(cp_last_date(EndDate)),
        ccheck_plan(T).
192 ccheck_plan([]).
    ccheck_plan([["-", _, "-"]|T]) :-
194     !,ccheck_plan(T).
    ccheck_plan([[StartDate, ID, EndDate]|T]) :-
196     cp_last_date(LastDate),

198     dateBeforeDate(LastDate, StartDate),

200     retractall(cp_last_date(_)),
        assert(cp_last_date(EndDate)),
202
        ccheck_plan(T).
204
    % == planning related stuff (replanning) ====================================
206
    % checks if plan is valid, if necessary creates some new one
208 validate_replan() :-
        current_plan(Plan),
210     check_plan(Plan),
        !,
212     debug("old plan still holds!").
    validate_replan() :-
214     debug("old plan is no longer possible, building a new one ..."),
        debug(""),
```

```
216     retractall(current_plan(_)),
        best_plan(Plan, Sum),!,
218     assert(current_plan(Plan)),
        debug("here is the new plan:"),
220     print_plan(Plan, Sum).
    validate_replan() :-
222     debug("sry, no other plan is possible").

224 % == planning related stuff (data retrieval) ================================

226 % asks the flight agency for the current set of flights
    get_all_flights() :-
228     flight_agent(To),
        debug(["querying ", To, " for all available flights ..."]),
230     sendMsg(ID, jade, To, query, flights()),
        rcvMsg(ID, jade, From, inform, [flights,Flights|_]),
232     debug("... got response"),
        retractall(flight(_,_,_,_,_,_,_)),
234     assert_flights(Flights).

236 % transforms the result list into some facts
    assert_flights([]).
238 assert_flights([[ID, From, To, FromDate, ToDate, Class, Costs]|R]) :-
        assert(flight(ID, From, To, FromDate, ToDate, Class, Costs)),
240     assert_flights(R).

242 % == main method ============================================================
    test() :-
244     get_all_flights(),
        best_plan(Plan, Sum),!,
246     assert(current_plan(Plan)),
        debug(""),
248     debug("here is the plan:"),
        print_plan(Plan, Sum),
250     Thread.sleep(3000),
        debug("starting the rearrangement simulation ..."),
252     debug("Simulation 1: change of flight; plan is still possible ..."),
        flight_agent(To),
254     sendMsg(ID, jade, To, query, simulate_flight_change_1()),
        debug("Simulation 2: change of flight; plan is no longer possible ..."),
256     sendMsg(ID2, jade, To, query, simulate_flight_change_2()),
        debug("Simulation 3: cancellation of some flight ..."),
258     sendMsg(ID3, jade, To, query, simulate_flight_cancel()).

260 % == auxiliary rules ========================================================

262 print_plan([], Costs) :-
        debug("==============================================="),
264     debug(["Overall price: ", Costs]),
        println([""]).
266 print_plan([[_,H|R]|T], Costs) :-
        print_item(H),
268     print_plan(T, Costs).

270 print_item(I) :-
        print_flight(I), !.
272 print_item(I) :-
        print_train(I), !.
274 print_item(I) :-
        print_hotel(I), !.
276
    print_hotel(ID) :-
278     hotel(ID, City, Name, Stars, _, _, Costs),
        debug([" Hotel: ", Name, " (", Stars, " stars, ", Costs, " Euro)"]).
280
    print_flight(ID) :-
282     flight(ID, From, To, FDate, TDate, Class, Price),
        debug(["Flight ", ID, ": ", From, " (", FDate, ") -> ", To, " (",
284         TDate, ") (", Class, ", ", Price, " Euro)" ]).
```

```
286 print_train(ID) :-
        train(ID, From, To, FDate, TDate, Class, Price),
288     debug(["Flight ", ID, ": ", From, " (", FDate, ") -> ", To, " (",
            TDate, ") (", Class, ", ", Price, " Euro)" ]).
290
    concatLists([], L2, L2).
292 concatLists([H|T], L2, [H|R]) :-
        concatLists(T, L2, R).
294
    elementOfList(X, [X], []) :- !.
296 elementOfList(X, [X|R], R).
    elementOfList(X, [F|R], [F|R2]) :-
298     elementOfList(X, R, R2).

300 lastElement(X, [X]) :- !.
    lastElement(X, [_|T]) :-
302     lastElement(X, T).

304 cityFromId(ID, City, Date) :-
        flight(ID, _, City, _, Date, _, _).
306 cityFromId(ID, City, Date) :-
        train(ID, _, City, _, Date, _, _).
308
    lastElementsCity(City, Date, L) :-
310     lastElement([_, ID|], L),
        cityFromId(ID, City, Date).
312

314 dateToDMYHM(Date, Day1, Month1, Year1, Hour1, Minute1) :-
        tokenize_list(Date, "/", [Datum, Time|_]),
316     tokenize_list(Time, ":", [Hour, Minute|_]),
        tokenize_list(Datum, "-", [Day, Month, Year|_]),
318     toInt(Hour, Hour1),
        toInt(Minute, Minute1),
320     toInt(Day, Day1),
        toInt(Month, Month1),
322     toInt(Year, Year1).

324 dMYHMToDate(Day, Month, Year, Hour, Minute, Date) :-
        implode(":", [Hour, Minute], Time),
326     implode("-", [Day, Month, Year], Datum),
        implode("/", [Datum, Time], Date).
328
    add(Num1, Num2, Result) :-
330     toInt(Num1, A),
        toInt(Num2, B),
332     Result = A + B.

334 toInt(Integer.Val, Integer.Val) :-
        !.
336 toInt(Val, Integer.Res) :-
        Res = Integer.parseInt(Val),
338     !.

340 egt(Integer.Val1, Integer.Val2) :-
        CV = Val1.compareTo(Val2),
342     CV >= 0.

344 lt(Integer.Val1, Integer.Val2) :-
        CV = Val1.compareTo(Val2),
346     CV < 0.

348 elt(Integer.Val1, Integer.Val2) :-
        CV = Val1.compareTo(Val2),
350     CV <= 0.

352 eq(Integer.Val1, Integer.Val2) :-
        CV = Val1.compareTo(Val2),
354     CV = 0.
```

```
addDaysToDate(Date, Days, NewDate) :-
    dateToDMYHM(Date, Day, Month, Year, Hour, Minute),
    NewDay = Day + Days,
    dMYHMToDate(NewDay, Month, Year, Hour, Minute, NewDate).

dateBeforeDate("-", _) :-
    !.
dateBeforeDate(_, "-") :-
    !.
dateBeforeDate(Date1, Date2) :-
    dateToDMYHM(Date1, Day1, Month1, Year1, Hour1, Minute1),
    dateToDMYHM(Date2, Day2, Month2, Year2, Hour2, Minute2),
    eq(Day1, Day2),
    eq(Month1, Month2),
    eq(Year1, Year2),
    !,
    elt(Hour1, Hour2),
    elt(Minute1, Minute2).
dateBeforeDate(Date1, Date2) :-
    dateToDMYHM(Date1, Day1, Month1, Year1, Hour1, Minute1),
    dateToDMYHM(Date2, Day2, Month2, Year2, Hour2, Minute2),
    elt(Day1, Day2),
    elt(Month1, Month2),
    elt(Year1, Year2).

sameDate(Date1, Date2) :-
    dateToDMYHM(Date1, Day1, Month1, Year1, Hour1, Minute1),
    dateToDMYHM(Date2, Day2, Month2, Year2, Hour2, Minute2),
    eq(Day1, Day2),
    eq(Month1, Month2),
    eq(Year1, Year2),
    eq(Hour1, Hour2),
    eq(Minute1, Minute2).

% two rules for the debugging output. the 1st takes lists of strings and the
% 2nd works with simple strings
% params:   in:LIST     list of strings you want to print out
debug([S|R]) :-
    !,
    concat(R,T),
    concat([S,T],F),
    debug(F).
% params:   in:String   the text you want to print out
debug(S) :-
    println(["[DEBUG] ",S]).
% params:   in:String   the text you want to print out
debugl(S) :-
    print(["[DEBUG] ",S]).

Output:
Travel Agent ----
Supported protocol 1: jade.mtp.http.MessageTransportProtocol
Launching a whole in-process platform...(Profile main=true local-host=bioinf3
port=1101 ...
     ...services=jade.core.mobility.AgentMobilityService;jade.core.event.NotificationService
host=bioinf3 local-port=1101 mtps=jade.util.leap.ArrayList@1e4457d jvm=j2se)
    This is JADE 3.2 - 2004/07/26 13:41:05
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/
http://bioinf3:7778/acc
Agent container Main-Container@JADE-IMTP://bioinf3 is ready.
[11:04:35] Adding node <Container-1> to the platform
[11:04:35] --- Node <Container-1> ALIVE ---
got request for flights
[DEBUG] simulating plan change ...
[DEBUG] simulating plan change ...
[DEBUG] simulating plan cancellation
-------------------------------------------------------------------------------------
Customer  ---
   This is JADE 3.2 - 2004/07/26 13:41:05
```

```
    downloaded in Open Source, under LGPL restrictions,
426    at http://jade.cselt.it/
   Agent container Container-1@JADE-IMTP://bioinf3 is ready.
428 Supported protocol 1: jade.mtp.http.MessageTransportProtocol(http://bioinf3:7778/acc)
   [DEBUG] querying flight@bioinf3 for all available flights ...
430 [DEBUG] ... got response
   [DEBUG]
432 [DEBUG] here is the plan:
   [DEBUG] Flight fl003: Munich (5-3-2005/10:00) -> Rome (5-3-2005/12:00) (economy, 60 Euro)
434 [DEBUG]  Hotel: Rome In (3 stars, 20 Euro)
   [DEBUG] Flight tr001: Rome (8-3-2005/11:11) -> Venice (8-3-2005/13:12) (economy, 10 Euro)
436 [DEBUG]  Hotel: Hilton Venice (5 stars, 70 Euro)
   [DEBUG] Flight tr002: Venice (9-3-2005/13:21) -> Milian (9-3-2005/16:16) (economy, 12 Euro)
438 [DEBUG]  Hotel: Milian-In (3 stars, 20 Euro)
   [DEBUG] Flight fl002: Milian (12-3-2005/14:45) -> Florence (12-3-2005/17:00) (economy, 30...
       ... Euro)
440 [DEBUG]  Hotel: Continental Florence (4 stars, 30 Euro)
   [DEBUG] Flight fl004: Florence (15-3-2005/09:30) -> Munich (15-3-2005/12:00) (economy, 65...
       ... Euro)
442 [DEBUG] ================================================
   [DEBUG] Overall price: 317
444 [DEBUG] starting the rearrangement simulation ...
   [DEBUG] Simulation 1: change of flight; plan is still possible ...
446 [DEBUG] Simulation 2: change of flight; plan is no longer possible ...
   [DEBUG] Simulation 3: cancellation of some flight ...
448 [DEBUG] something in flight fl002 has changed ...
   [DEBUG]   flight No. fl002
450 [DEBUG]      was: [DEBUG] Flight fl002: Milian (12-3-2005/14:45) -> Florence ...
       ...(12-3-2005/17:00) (economy, 30 Euro)
   [DEBUG]      is:  [DEBUG] Flight fl002: Milian (12-3-2005/14:55) -> Florence ...
       ...(12-3-2005/17:20) (economy, 30 Euro)
452 [DEBUG] old plan still holds!
   [DEBUG] something in flight fl002 has changed ...
454 [DEBUG]   flight No. fl002
   [DEBUG]      was: [DEBUG] Flight fl002: Milian (12-3-2005/14:55) -> Florence ...
       ...(12-3-2005/17:20) (economy, 30 Euro)
456 [DEBUG]      is:  [DEBUG] Flight fl002: Milian (16-3-2005/15:23) -> Florence ...
       ...(16-3-2005/18:00) (economy, 46 Euro)
   [DEBUG] old plan is no longe possible, building a new one ...
458 [DEBUG] here is the new plan:
   [DEBUG] Flight fl003: Munich (5-3-2005/10:00) -> Rome (5-3-2005/12:00) (economy, 60 Euro)
460 [DEBUG]  Hotel: Rome In (3 stars, 20 Euro)
   [DEBUG] Flight tr001: Rome (8-3-2005/11:11) -> Venice (8-3-2005/13:12) (economy, 10 Euro)
462 [DEBUG]  Hotel: Hilton Venice (5 stars, 70 Euro)
   [DEBUG] Flight tr002: Venice (9-3-2005/13:21) -> Milian (9-3-2005/16:16) (economy, 12 Euro)
464 [DEBUG]  Hotel: Milian-In (3 stars, 20 Euro)
   [DEBUG] Flight fl005: Milian (12-3-2005/17:42) -> Florence (12-3-2005/19:40) (economy, 35...
       ... Euro)
466 [DEBUG]  Hotel: Continental Florence (4 stars, 30 Euro)
   [DEBUG] Flight fl004: Florence (15-3-2005/09:30) -> Munich (15-3-2005/12:00) (economy, 65...
       ... Euro)
468 [DEBUG] ================================================
   [DEBUG] Overall price: 322
470 [DEBUG] flight No.fl004 was cancelled, ...
   [DEBUG] old plan is no longe possible, building a new one ...
472 [DEBUG] here is the new plan:
   [DEBUG] Flight fl003: Munich(5-3-2005/10:00) -> Rome (5-3-2005/12:00) (economy, 60 Euro)
474 [DEBUG]  Hotel: Rome In (3 stars, 20 Euro)
   [DEBUG] Flight tr001: Rome (8-3-2005/11:11) -> Venice (8-3-2005/13:12) (economy, 10 Euro)
476 [DEBUG]  Hotel: Hilton Venice (5 stars, 70 Euro)
   [DEBUG] Flight tr002: Venice (9-3-2005/13:21) -> Milian (9-3-2005/16:16) (economy, 12 Euro)
478 [DEBUG]  Hotel: Milian-In (3 stars, 20 Euro)
   [DEBUG] Flight fl005: Milian (12-3-2005/17:42) -> Florence (12-3-2005/19:40) (economy, 35...
       ... Euro)
480 [DEBUG]  Hotel: Continental Florence (4 stars, 30 Euro)
   [DEBUG] Flight fl006: Florence (15-3-2005/10:30) -> Munich (15-3-2005/13:00) (economy, 95...
       ... Euro)
482 [DEBUG] ================================================
   [DEBUG] Overall price: 352
```