INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Diplomarbeit

Reactivity on the Web: Event Queries and Composite Event Detection in XChange

Michael Eckert

Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Prof. Dr. François Bry
	Paula-Lavinia Pătrânjan
Abgabetermin:	20. Mai 2005

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst habe, und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den

(Unterschrift des Kandidaten)

Abstract

Reactivity, the ability to detect events and respond automatically in a timely manner, is an essential requirement in many present-day information systems. The reactive, rule-based language *XChange* aims at filling the gap between the current, largely passive Web and the need for reactivity.

XChange programs consist of rules, and each rule specifies events — happenings at (possibly remote) Web sites — it reacts upon using an *event query*. Event queries in XChange not only specify situations in which a rule reacts, but also extract and make available data from events that is relevant for the particular reaction. Oftentimes, the situations are not given by a single (atomic) event, but a series of events, leading to the notion of *composite events* and *composite event queries*.

This work is dedicated to the design of XChange's event query language, with an emphasis on the ability to query composite events. In this work, a number of language constructs are provided that allow to combine atomic event queries into composite event queries. Their intuitive meaning is underpinned with formal, declarative semantics for both atomic and composite event queries. By design, composite queries need only events with a bounded life-span to be evaluated, and a proof of this based on the declarative semantics is offered in this work. Algorithms for the evaluation of event queries, in particular the detection of composite events, are provided and accompanied by a prototype implementation.

Zusammenfassung

Reaktivität, die Fähigkeit Ereignisse zu erkennen und schnell auf sie zu reagieren, ist eine wesentliche Anforderung an viele heutige Informationssysteme. Die reaktive, regelbasierte Sprache *XChange* füllt die Lücke zwischen dem gegenwärtigen, hauptsächlich passiven Web und der Erfordernis von Reaktivität.

XChange Programme bestehen aus Regeln, die auf Ereignisse an (möglicherweise entfernten) Web Sites reagieren. Die Ereignisse, auf die eine Regel reagiert, werden mittels einer *Ereignisanfrage* (event query) angegeben. Ereignisanfragen dienen aber nicht nur dazu, Situationen zu spezifizieren in denen eine Regel reagiert. Sie dienen auch dazu, Daten aus den Ereignissen abzufragen und zur Verfügung zu stellen, die relevant für die jeweilige Reaktion sind. Oft lassen sich Situationen nicht durch ein einziges (atomares) Ereignis beschrieben, sondern nur durch eine Abfolge von Ereignissen, was zum Begriff der *zusammengesetzten Ereignisse* (composite events) und zu *zusammengesetzten Ereignisanfragen* führt.

Diese Arbeit entwirft die Ereignisanfragesprache für XChange, mit Betonung auf die Möglichkeit, zusammengesetzte Ereignisse anzufragen. Verschiedene Sprachkonstrukte werden in dieser Arbeit vorgestellt, die es erlauben, zusammengesetzte Ereignisanfragen aus atomare Ereignisanfragen zu bauen. Die intuitive Bedeutung der Sprachkonstrukte wird durch eine formale, deklarative Semantik für atomare und zusammengesetzte Ereignisanfragen untermauert. Die Sprache ist so entworfen, dass für die Auswertung von zusammengesetzten Ereignisanfragen nur Ereignisse mit einer begrenzten Lebensdauer benötigt werden. Basierend auf der deklarativen Semantik wird dies in dieser Arbeit auch formal bewiesen. Begleitet von einer Prototyp-Implementierung werden Algorithmen zur Auswertung von Ereignisanfragen, insbesondere zur Erkennung von zusammengesetzten Ereignissen, vorgestellt.

Acknowledgments

The work presented in this thesis would not be what is today without the support and contribution of many people. Foremost, I would like to express my gratitude to my thesis supervisors *François Bry* and *Paula-Lavinia Pătrânjan*, both of the University of Munich, for working with me and for the many fruitful discussions. Also, I would like to thank *James Bailey*, University of Melbourne, for his feedback on this work during his collaboration with my thesis supervisors and me on a research article on event queries in XChange.

Many people in the *programming and modeling languages research group* at the University of Munich have provided me with help during this thesis. Together they have created an excellent research environment that I am very grateful for.

For proof-reading and reading this thesis with the eyes of a someone involved in a different research area of computer science, I thank my colleague and friend *Jochen Wuttke*.

Research in this work has been partly funded by the *European Commission* and by the *Swiss Federal Office for Education and Science* within the 6th Framework Programme project REWERSE number 506779 (see `http://rewerse.net`). I am thankful for the funding that enabled me to attend workshops and conferences that helped improve this work.

I am indebted to the *Studienstiftung des deutschen Volkes* for its financial and non-financial support throughout the course of my graduate studies.

I am also indebted to the *German-American Fulbright Commission* for its financial and non-financial support of my studies at the University of Washington in the year 2002/03.

Last but not least I thank my family, especially my parents, for continously supporting and encouraging me.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goals and Contributions 2

1.3 Organization of this Thesis 3

2 Paradigms and Design-Decisions of XChange **5**

2.1 Basic Architecture 5

2.1.1 Local XChange Programs 5

2.1.2 Reactivity to Events 5

2.1.3 Support for Composite Events 6

2.1.4 XChange Programs as Event-Condition-Action Rules 6

2.2 Querying Data and Reasoning 6

2.2.1 Data in Events 6

2.2.2 Data in Web Resources 7

2.2.3 Clear Separation of Volatile and Persistent Data 7

2.2.4 Bounded Event Life-Span 8

2.3 Communication between Web Sites with XChange Programs 9

2.3.1 Push Communication of Events 9

2.3.2 Asynchronous Communication 11

2.3.3 Peer-to-Peer Communication 11

2.4 Evolution of Data 11

2.4.1 Local and Remote Updates 11

2.4.2 Pattern-based Updates 12

2.4.3 Complex Updates 12

2.4.4 Support for Transactions 12

2.4.5 Global Evolution through Reactivity 13

3 An Overview of the Language XChange **15**
3.1 Xcerpt: A Web Query Language . . . 15
3.1.1 Data Terms . . . 16
3.1.2 Construct-Query Rules and Goals . . . 19
3.1.3 Query Terms and Queries . . . 20
3.1.4 Construct Terms . . . 22
3.1.5 A Visual Query Language based on Xcerpt: visXcerpt . . . 24
3.1.6 Summary . . . 24
3.2 XChange: A Reactive Language for the Web . . . 26
3.3 Events in XChange . . . 26
3.3.1 Representation of Atomic Events . . . 27
3.3.2 Purpose of Event Queries . . . 27
3.3.3 Atomic Event Queries . . . 27
3.3.4 Composite Event Queries . . . 29
3.4 Conditions in XChange . . . 30
3.4.1 Web Queries as Conditions . . . 30
3.4.2 Deductive Rules in XChange Programs . . . 30
3.5 Actions in XChange . . . 32
3.5.1 Raising Events . . . 34
3.5.2 Transactions . . . 35
3.5.3 Elementary Updates: Update Terms . . . 35
3.5.4 Complex Updates . . . 38
3.6 Summary . . . 40

4 Event Queries in XChange **43**
4.1 Notes on the Terminology . . . 43
4.2 Atomic Events . . . 44
4.3 Querying Atomic Events . . . 47
4.4 Answers to Atomic Event Queries . . . 49
4.5 Composite Events . . . 50
4.6 Answers to Composite Event Queries . . . 50
4.7 Composite Event Queries . . . 53
4.7.1 A First Set of Operators: Conjunction, Disjunction, Sequence . . . 54
4.7.2 Temporal Restrictions . . . 56
4.7.3 Variables . . . 58

4.7.4 Conditions 60
4.7.5 More Operators 60
4.8 Legal Event Queries 64

5 Declarative Semantics for Event Queries 67
5.1 Atomic Events and Event Stream 68
5.2 Answers to Event Queries 69
5.3 Relating Queries and Answers 71
5.3.1 Atomic Query 72
5.3.2 Conjuction 73
5.3.3 Disjunction 73
5.3.4 Sequence 74
5.3.5 Absolute Temporal Restriction 75
5.3.6 Relative Temporal Restriction 75
5.3.7 Variable Restriction 75
5.3.8 Exclusions 76
5.3.9 Quantifications 76
5.3.10 Multiple Inclusions and Exclusions 77

6 Legal Event Queries and Bounded Event Life-Span 79
6.1 Reminder: Definition of Legal Event Queries 79
6.2 Theorem: Evaluation with Bounded Life-Span 80
6.3 Lemma 1: Bound for the Answers to Legal Event Queries 80
6.4 Lemma 2: Restriction of the Event Stream 81
6.5 Discussion 84

7 Incremental Evaluation of Event Queries 85
7.1 Requirements and Considerations 85
7.2 Evaluation of Atomic Event Queries 87
7.3 Evaluation of Composite Event Queries 87
7.3.1 Related Work on Composite Event Detection 88
7.3.2 Composite Event Detection in XChange 89
7.3.3 Tree-Based Representation of Partial Event Query Evaluations 90
7.3.4 Bottom-Up Data-Flow for Event Detection 91
7.3.5 Top-Down Traversal for Event Deletion 93

7.3.6 Special Considerations: Duplicate Elimination, Partial Matches, Negation, Existential Quantification . . . 95
7.4 Correctness of the Incremental Event Query Evaluation . . . 98
7.5 Ideas for Optimizations . . . 98

8 Implementation **103**
8.1 An Architecture for XChange . . . 104
8.2 Event Reception and Handling . . . 105
8.3 Event Query Evaluation . . . 106
8.4 Event Deletion . . . 107
8.5 Instructions for Building and Running XChange . . . 107
8.6 Unimplemented Aspects and Future Work . . . 109

9 Conclusions **111**
9.1 Summary . . . 111
9.2 Related Work . . . 111
9.3 Future Research Directions . . . 112
9.3.1 Language Design . . . 112
9.3.2 Semantics . . . 113
9.3.3 Event Query Evaluation . . . 114
9.4 Conclusion . . . 115

A Grammar for XChange Event Queries **117**
A.1 EBNF notation . . . 117
A.2 Event Queries . . . 118
A.3 Legal Event Queries . . . 119
A.4 Time Specifications . . . 119

B Document Type Definition for Event Messages **121**

C Notation used for Semantics **123**
C.1 Atomic Events . . . 123
C.2 Substitution Sets . . . 123
C.3 Event Sequences . . . 124
C.4 Stream of Incoming Events . . . 124
C.5 Operations on Event Sequences . . . 124
C.6 Relating Queries and Answers . . . 125

D On Event Sequences as Answers **127**

Bibliography **129**

List of Figures

3.1 Data term representation of an HTML document 16
3.2 Data term representation of XML or other tree-structured data 17
3.3 Graph-structured data 18
3.4 Xcerpt rule 23
3.5 VisXcerpt rendering of an Xcerpt rule 25
3.6 Syntax of ECA rules in XChange 26
3.7 Representation of an Event 28
3.8 An atomic event query in XChange 29
3.9 A Web query in XChange 31
3.10 Excerpt of the flight database `lufthansa.xml` 31
3.11 Excerpt of the flight database `united.xml` 32
3.12 An XChange program using rule chaining 33
3.13 An Event raising specification in XChange 34
3.14 An elementary update in XChange 36
3.15 Greedy vs. parsimonious deletion in a graph 37
3.16 A complex update in XChange 39
3.17 Another complex update in XChange 40

4.1 Data term representation of a sequence of atomic events 52

7.1 A (composite) event query and its operator tree 90
7.2 Operator tree for a `times...during...`-query 94
7.3 Optimization of an operator tree into an operator graph 100
7.4 Optimization of composite event queries with shared sub-expressions 100

8.1 The architecture of the XChange implementation 105

Chapter 1

Introduction

Reactivity, the ability to detect events and respond automatically in a timely manner, is an essential requirement in many present-day information systems. 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 — hosted 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 on the Web (*change*) and event-based communication between Web-site (*exchange*), hence also its name, XChange.

XChange programs react upon *events*, happenings at Web sites. To specify situations that require a reaction, XChange provides *event queries*, which describe classes of events. But event queries do more: they also extract and make available data from events that is relevant for the particular reaction. Oftentimes, the situations are not given by a single event, but a series of events, leading to the notion of *composite events* and *composite event queries*. This work designs and implements querying events and detecting composite events as part of the XChange language.

1.1 Motivation

In XChange's reactive setting, Web sites react upon events happening either locally or at other, remote Web sites. An event is some change in the world's state; this includes, but is not limited to, 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." When an event happens at some Web site, the Web site communicates this to other, potentially interested Web sites.

The information about events that flows between Web sites will be called an event message. In XChange, event messages have XML syntax and are communicated between Web sites in a push-manner.

Similar, limited forms of this reactivity, we can already find on today's Web: some Web sites (e.g., news services) allow a user to register interest in a particular Web page, so that when the Web page changes he will automatically be sent an e-mail informing him about the change. For the user, this can be quite convenient: he does not have to check the Web page periodically for changes. For the server and the network infrastructure, this is convenient, too: query load at the server and network traffic are significantly reduced.

XChange introduces an important distinction concerning data on the Web. On the one hand we have the actual Web data, that is, data contained in Web resources such as HTML and XML documents. Web data is stored persistently and can be retrieved upon request in a pull-manner. On the other hand we have event data, that is, data informing us about changes of Web data (and other changes in the world's state). The event data is valuable only temporarily: it serves only the purpose of informing us so we can decide whether and how to react to the change. Event data is of volatile nature and communicated in a push-manner. XChange is carefully designed to provide a clear separation between persistent and volatile data. Without this clear separation, it would be unclear for the programmer which data is "normal" Web data and which is event data. Also, it would be unclear for the language processor when events can be discarded, and thus lead to a memory demands that increase over time.

Issues concerning reactive information systems have been explored in active database research for quite some while. However, the Web differs significantly in its architecture from usual active databases. XChange and in particular the event query language of XChange presented in this work are designed to take into consideration Web specifics such as loosely structured (or semi-structured) data, unreliable communication, openness, and decentralization.

1.2 Goals and Contributions

This work has the following goals:

- design, refine and clarify the syntax and semantics of XChange's event query language based on preliminary work in [BBP04],
- provide the event query language with declarative semantics,
- develop an algorithm to evaluate (composite) event queries in an incremental fashion, and
- implement a running prototype of the incremental event query evaluation.

This work builds upon and extends related work concerned with composite events (mainly from the active database community) but also differs significantly in some aspects. The most important contributions can be summarized as follows:

- Atomic events are not limited to a fixed set of types but can be arbitrary, semi-structured data (XML documents etc.).

- XChange's event query language serves not only to specify (atomic and composite) events but also to extract data contained in the queried events. This data can influence the reaction to a detected event.
- XChange comes with a notion of legal event queries that can (as proven in this work) be evaluated with events having only a bounded life-span. Bounded event life-spans are important to avoid a monotonous increase in event storage requirements over time.
- Formal semantics for the event query language have been developed. The chosen approach can accommodate advanced features such as free variables in queries, event negation, and partial matches.
- Influenced by previous work in active database research, an algorithm to evaluate XChange's event queries incrementally has been devised and implemented. The algorithm handles language features such as variables, event negation, and partial matches, and also deletion of events dispensable due to "expiration" of their life-span.
- Since XChange's event query language expressly considers data contained in events, formal semantics and incremental evaluation are more complicated than for many previous composite event query languages. The increased complexity is due to the presence of (logical) variables in event queries. On the other hand, the complex semantics give rise to a simple programming.

In a much shorter form, we also discuss the main points of this thesis in [BBEP05].

1.3 Organization of this Thesis

After this introduction (Section 1), we present the paradigms and design decisions that provide the basis for work on XChange and its event query language in particular (Section 2). The overview of the language XChange as a whole (Section 3) is followed by an introduction of XChange's event query language (Section 4). Next, we define formal semantics for the event query language (Section 5). From the semantics we prove that, by the language's design, all legal event queries can be evaluated with events of bounded life-span (Section 6). We then present an algorithm for the incremental evaluation of event queries (Section 7). This is followed by an overview of its prototype implementation (Section 8). Conclusions and a discussion of future work round off this thesis (Section 9).

Chapter 2

Reactivity on the Web: Paradigms and Design-Decisions of XChange

This chapter presents paradigms and design-decisions upon which XChange, a language for reactivity and evolution on the Web, is built. The driving force in establishing the paradigms and design-decisions are the characteristics of the Web that have to be taken into account.

This chapter is organized to first give a very broad view of paradigms and decisions governing the design of XChange, and subsequently go into further detail.

2.1 Basic Architecture

2.1.1 Local XChange Programs

The Web is built as a completely decentralized system. It is a loose collection of *resources* that are identifiable by *Uniform Resource Identifiers* (URIs, [BFM98]) or, in near future, *International Resource Identifiers* (IRIs, [DS04]). A group of resources that underly a common administration, which usually means they are hosted on the same Web server, is often called *Web site*.

In order to not violate the decentralization, XChange programs run *locally* at a Web site— called an *XChange-aware Web site*— and can modify directly only local resources. Global behavior comes only from communication between local XChange programs. To support communication, XChange programs are addressable at (one or more) URIs.

2.1.2 Reactivity to Events

XChange programs are reactive: when a certain event happens, they respond automatically by executing some previously specified actions.

An event is some occurrence of interest, a change in the state of the world, to which a Web site can react in a particular way or not react at all. Abstraction levels of events vary. For example, the modification of a document on the Web is an event with a low abstraction level, while the cancellation of a flight is an event with a high abstraction level. XChange provides

a set of basic, low-level events such as insertion, deletion, update of an element in a local XML document, timer events, system events, and so on. Application-dependent, higher-level events can be constructed (raised) from the basic events or combinations of basic events.

An XChange-aware Web site locally processes all *incoming events*, i.e., all events it is notified of, and checks for each particular incoming event if the XChange program requires a reaction.

2.1.3 Support for Composite Events

There are cases where a situation that requires a reaction cannot be inferred from a single event (called *atomic event* henceforth). For example, the cancellation of a flight (atomic event) might not by itself require a reaction by a passenger. However, if a flight has been canceled, and there is no notification within the next two hours that the passenger is put onto another flight, this requires a reaction.

To accommodate such situations, XChange supports so-called *composite events*. Composite events are situations of interest that cannot be inferred from the occurrence of a single atomic event, but need to detect the occurrence or non-occurrence of several atomic events. Composite events have, unlike atomic events, a duration; they stretch over a time interval covering at least the occurrence times of all their constituting atomic events.

2.1.4 XChange Programs as Event-Condition-Action Rules

A natural candidate to implement reactive functionality are *Event-Condition-Action* (ECA) rules. An ECA rule specifies that some *action* is performed automatically in response to an *event*, provided that the *condition* holds. An XChange program is a set of *Event-Condition-Action* (ECA) rules. (As we will see later, an XChange program may additionally contain deductive rules to support advanced reasoning capabilities.)

ECA rules have been successfully employed in many domains requiring reactive functionality [BPW02], e.g., to implement constraints enforcement in databases, to maintain data warehouses, in active databases, in work-flow management, in network management, and in specifying and implementing business processes.

Due to their highly declarative nature, ECA rules are easily analyzed. This is especially true in comparison to implementing reactive functionality in conventional programming languages such as Java, C, or C++, where already non-reactive programs can be hard to analyze.

2.2 Querying Data and Reasoning

2.2.1 Data in Events

The notion of what constitutes an event is application-dependent. Events in relational database systems are often limited to *insertion*, *deletion*, and *update* of a data record.

In contrast, XChange is intended to cover a variety of application-dependent events. These include basic, domain-independent data events such as insertion, deletion, and updates, but can also be domain-dependent events such as the cancellation of a flight. Hence, it does not

make sense to limit the types of events before-hand. Instead, events in XChange are represented as semi-structured XML data. This gives the application programmer all flexibility to choose the necessary types of events and good representations.

The representation of an event is called an *event message*; however since we assume that every relevant event is given a representation, we will use the terms event and event message often synonymously.

Events (or better their message representations) contain data that can be necessary to decide on an appropriate reaction. For example, an event for insertion of a data item should include the location where the data was inserted and maybe also the inserted data itself. An event for the cancellation of flight should include information about the flight such as airline, flight number, departure date, and maybe also a list of the affected passengers.

The event-part of an XChange ECA-rule needs means to specify classes of events —atomic or composite— the rule reacts to (fires), and means to extract the data that influences the reaction. This is done using event queries, which constitute this work's main contribution to XChange.

To the best of our knowledge, XChange is currently the only language with support for composite events that expressly takes into account data contained in events.

2.2.2 Data in Web Resources

The appropriate reaction to some event depends usually not only on the data provided by the event, but also on data contained in arbitrary Web resources (called *Web data* for short) such as XML documents available locally or anywhere on the Web. In contrast to events that are sent to an XChange program in a push-manner and discarded after some time, data in Web resources is persistent and has to be retrieved in a pull-manner.

To access persistent Web resources and extract data from them, XChange incorporates the query language Xcerpt [Sch04]. With Xcerpt, one can query tree- and graph-structured data such as XML and RDF. To incorporate advanced reasoning capabilites that make XChange also suitable for Semantic Web applications, XChange programs can contain deduction rules from Xcerpt in addition to the reactive ECA rules.

Note that XChange has an emphasis on reactivity, so the way chosen is to *incorporate* a query language into the reactive framework. The other way round, *enhancing* some given query language with reactive capabilities, might be appropriate in settings where the focus is on querying and deduction but reactivity plays only a minor role.

2.2.3 Clear Separation of Volatile and Persistent Data

In a reactive Web there are two kinds of data: data from events (event data) and data from Web resources (Web data). For both kinds, data sources are in XML format (or a derived format such as RDF), and the same query language can —and for user's convenience should— be used to extract data from them.

However, Web data and event data differ in their nature. 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 received in a *push*-manner. Also, in composite events, *temporal patterns* of events play a role.

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 by the author). 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, they only way to correct, complete, or invalidate its information then is through speaking new sentences.

Due to their different nature, there should be a clean separation of persistent Web data and volatile event data in a reactive language. It should be ensured that volatile data stays volatile, i.e., is disposed of after finite time. This avoids growing storage requirements for event data. If some data from an event must be stored indefinitely, it should explicitly be made persistent in a Web resource. Most importantly, not having a clean separation of Web data and event data could lead to a "shadow-Web", a hidden collection (then non-volatile) of event data that lives in parallel to the normal Web with its persistent Web resources.

If some data that arrives in events —volatile data— is needed permanently, it can and should be turned into persistent data: A rule extracts the relevant data from the event message (event-part) and updates a persistent Web resource to store the data (action-part), turning the data from volatile event data into persistent Web data. Note that event messages include all book-keeping information (sender of the message, reception-time, etc.), so this information can be made accessible when turning the event into persistent data and does not get lost.

For the programmer not to confuse data of volatile and persistent nature, and for the language processor to be able to recognize easily discardable event data, XChange strives for a clear separation of access to event data and Web data: data from events can only be extracted in the event-part of the ECA-rules, while data from Web resources can only be extracted in the condition-part of the ECA-rules. Of course the Web query posed in the condition-part can depend on data extracted previously in the event-part by an event query (but not vice-versa!). Variable assignments allow a flow of data from the event-part to the condition-part to the action-part.

2.2.4 Bounded Event Life-Span

To keep volatile event data volatile and to keep storage requirements constant, every event must have a limited life-span and must be disposed of after a bounded time.

Whether some given event can disposed of, depends on the event queries: an event must only be kept in memory if it is needed by some query. The event query language in XChange is designed in a way that every event can be disposed of after a bounded time. An upper bound on the life-span of every event can be determined statically just from the event queries; no knowledge of the actual dynamic run-time behavior (such as the sequence of events happening) is needed. XChange captures this important restriction on event queries in the notion of *legal* event queries.

The set of event queries present at some XChange-aware Web-site is however not static: new ECA-rules (containing new event queries) can be registered at any time. Since a new event

query may arrive at any time in the future, event queries are not allowed to look into their past. More precisely: an event query is evaluated only against events happening *after* the rule it is part of has been registered at the XChange-aware Web-site. Any events that happened before that time are irrelevant to this query.

For atomic event queries, things are now simple. All atomic event queries are legal, because atomic events used to answer an atomic event query can be discarded immediately and thus do not really need a life-span at all: Every incoming atomic event is tested against each atomic event query. If it answers the atomic event query, the associated condition is tested *immediately* and the associated action is executed (provided the condition test has been successful). After this, the atomic event is not needed anymore by any atomic event query.

For composite event queries, things are harder. Consider this example of a rule with a composite event query detecting sequences of (atomic) events `a` and `b`: "Inform my business colleagues of my new arrival time (action), when my flight to the meeting place has been canceled (atomic event `a`) and I received notification that I have been booked on an alternative flight (atomic event `b`)." Once event `a` has been received, the fact that `a` happened and maybe also data contained in `a` has to be kept in memory at least until `b` happens (potentially even longer). However, there is not guarantee that `b` actually will happen, so `a` might have to be kept in memory *forever*. Thus, such an event query is *not legal*.

To avoid such situations, all legal composite event queries in XChange have to specify explicitly a "time-out" that gives a maximum life-span to each event that is potentially part of an answer to a composite event query. In the example of a (not legal) event query from above, we can assume that the notification for an alternative flight comes shortly after the flight cancellation. This means that we have to state "shortly after" more precisely as something like "within one day." An implicit time-out, i.e., a time-out that is not specified separately in each query but given globally by the system, does not make much sense in the XChange setting, since the length of such a time-out is generally application-dependent.

The issue of a bounded life-span for events has first been raised in [BZBW95], but not been elaborated on much. To the best of our knowledge, XChange's event query language is currently the only language taking this into consideration and offering a *formal proof* (cf. Chapter 6) that a bounded life-span for events suffices to answer all legal event queries.

2.3 Communication between Web Sites with XChange Programs

2.3.1 Push Communication of Events

XChange programs running at different Web sites communicate through events, which are, as mentioned above, arbitrary messages represented in XML. XChange uses a *push* approach to communicate events. This is however not the only conceivable approach, and the decision to use it requires some further explanation and investigation of the alternatives.

Let us first describe the setting: a number of Web sites is interested in certain events occurring or being raised at some Web site. We will call the latter event *emitter* and the former event *observers* (or *recipients*).

In principle, two approaches exist that differ in who carries the burden of event notification or detection: *push* communication of messages informing about events and *periodical polling* of Web resources to check whether an event has happened.

In *push communication*, event messages are *sent* to all observers (Web sites interested in a certain event) by the emitter (the Web site where the event occurs or is raised). The observers are automatically notified of the event and have no effort. This requires that all observers are known to the emitter.

Alternatively, observers could *periodically poll* the emitter Web site to check if an event (e.g., a change in the Web site's data) has occurred. Event detection can be done in several ways, for example,

- by evaluating a query against the data at the emitter Web site (an event happened if the query is satisfied),
- by examining the difference between versions of the data (an event happened if the version from the previous polling differs to the current version), or
- by requiring the emitter Web site to publish a log of time-stamped events (an event happened if the emitter lists an event with a time-stamp later than the time of the last polling).

For reactivity on the Web, push communication seems to be better suited. In general, push communication causes less network traffic, allows faster reaction to events, and allows the emitter to initiate communication. Also, push communication gives a clearer separation of volatile and persistent data (see 2.2.3 above).

Network traffic. Push communication causes less network traffic since data is only sent when there actually happened some event. Polling, in contrast, requires data to be sent every polling interval even if no event happened.

Reaction time. Push communication makes observers immediately aware of some event, so they can react in very short time. Polling delays event detection at the observers until the next periodical poll. In average the observers have to wait for half the length of the polling interval.

Initiation of communication. Push communication allows an emitter to initiate communication, i.e., an emitter can send a message to some Web site which previously has not been among its observers. Pure polling does not allow this.

Separation of volatile and persistent data. When events are communicated in push manner, the clear separation of volatile and persistent data from above is also visible in the way the data is communicated. Volatile data, i.e., events, is communicated in a push manner, while persistent data, i.e., Web resources, is retrieved in a pull manner. Communicating events in a polling approach, volatile event data and persistent Web resource data are very

similar and easily confused. E.g., in the case where an emitter publishes a log of its events, this log can be seen as just another (persistent) Web resource.

Also note that periodical polling, if it is deemed necessary in some application, can be realized in XChange using periodical timer events. One can use a timer event to fire a rule periodically and use its condition part to access some Web data in a poll manner.

2.3.2 Asynchronous Communication

XChange-aware Web sites communicate by sending and receiving event messages. Sending of events is performed *asynchronously*. That is, the send-operation to communicate an event message to other Web sites is *non-blocking*. Execution of an XChange-program continues immediately without waiting for the transmission of the event message or waiting for a reply to the event message.

Asynchronous communication seems to be the most favorable for a reactive Web site. Using synchronous message-passing, execution of a program has to be suspended until a sent message is transmitted correctly. In a setting like the Web with unreliable communication, the suspension time can be quite long and has no clearly established upper bound. During this time the Web site cannot react to other incoming events.

2.3.3 Peer-to-Peer Communication

Communication of XChange-aware Web sites is based on a *peer-to-peer* model: all Web sites have the same capabilities and any Web site can initiate communication to another Web site (if the other Web site's URI is known). Event messages are exchanged directly between Web sites. No centralized instances, super-peers, or the like exist to control and synchronize communication. XChange stays consistent with the design of the Web as a *decentralized* system. Any centralized instances could cause severe scaling problems, and also non-technical issues, e.g., who provides the centralized instances, who has to pay their cost, and what is a suitable price structure for their usage.

It is not possible to broadcast an event message to all XChange-aware Web sites since broadcasting on the Web's infrastructure is completely out of question. All recipients of an event message have to be specified explicitly. If some Web site is interested in another Web site's events, it can register its interest and *subscribe* to relevant events. Event notification follows the *publisher-subscriber* model. Note however that a publishing Web site can initiate communication and send an event also to Web sites that have not subscribed to the event.

2.4 Evolution of Data

2.4.1 Local and Remote Updates

XChange programs react to certain events by automatically executing some appropriate action. Such an action can be sending an event message (*exchange* of data) or modifying persistent Web data (*change* of data). XChange programs can modify *local data* (available

locally at their Web site) as well as *remote data* (available at other XChange-aware Web sites).

Modifications of remote data, *remote updates*, is performed by sending a request to the remote Web site. Success of a remote update is not guaranteed; a Web site may refuse an update request due to lack of access rights or credentials, or the update may fail because of communication or system failures. As long as no communication or system failures take place, the remote Web site sends a response to the requesting Web site indicating success or failure of the update request.

In its present stage, XChange provides no means to deal with security issues such as authorization (does a another Web site have the rights to modify this data?), authentication (is the communication partner really who it claims to be?), and trust (if a Web site claims to have executed an update request, can other believe that and rely on it?). These issues are largely orthogonal to the current work on XChange, and extensions of XChange in this direction should pose no great difficulties.

2.4.2 Pattern-based Updates

The smallest and simplest form of an update in XChange is an *elementary update*. It is a change of a data item (e.g., an XML element or an RDF triple) within a persistent Web resource (e.g., an XML or RDF document).

Elementary updates are pattern-based: an elementary update is specified by a (possibly incomplete) pattern for the data to be updated, containing additionally the desired update operations. Note that such a pattern can contain more than one update operation, and these can have different types (i.e., insert, delete, replace).

2.4.3 Complex Updates

In real-life applications, it is often required to change a number of data sources simultaneously and in a consistent manner. Such an update is called a *complex update*, and it consists of a number of *dependent* elementary updates that need to be executed in a *synchronized* fashion. For example, one might want to book a flight *and* a hotel reservation, *or else* (if that failed) book a seat on the night train.

Complex updates in XChange are ordered or unordered conjunctions and disjunctions of (elementary and complex) updates. When executing a complex update, the relations between the elementary updates are taken into account and they are executed synchronously.

2.4.4 Support for Transactions

Transactions allow the execution of updates in an *all-or-nothing* manner: a complex update either completes successfully (at all Web sites it updates), or it has no effect at all. Transactions should obey the ACID properties [HR83]:

- Atomicity, a transaction either takes place as a whole or not at all;

- Consistency, a transaction takes the system from one consistent state to another consistent state;
- Isolation, each transaction must be performed without inference from other, concurrent transactions;
- Durability, after a transaction has completed successfully, all its effects are saved in the permanent storage.

Transaction management in distributed systems [CDK01] is a difficult issue, even in distributed systems of a smaller scale than the Web. The XChange language principally supports the concept of transactions, relying on the applicability of standard solutions from distributed systems research.

However, transaction management is not the current research focus of XChange, and the current transaction mechanisms are amendable. For a proof of concept design and implementation of XChange the current mechanisms should suffice, but significant further research might be required to make XChange fit for deployment in practical, real-life environments.

2.4.5 Global Evolution through Reactivity

Different Web sites often have a dependency on each others data and need to be kept in sync and consistent. For example, entries in a Web-based personal calendar of a business traveler depend on the arrival and departure times of trains, flights, etc., the traveler uses. If a flight is delayed, the calendar needs to be adapted and appointments such as business meetings have to be postponed or canceled. This in turn requires notification of the other meeting participants and further adaption of their calendars.

From this example we can see that the Web is not just a collection of independent data sources that evolve locally. Rather, it is a collection of information systems that depend on each other and exhibit global behavior. With this perspective, the Web becomes a "living organism" of information systems that evolve together globally.

Several attempts can be made towards a framework supporting *global evolution*. XChange supports global evolution through reactivity. A Web site that depends on another Web site's data can register interest in certain changes of the data and will be notified of changes. Upon such a notification, the dependent Web site can then react and change its own data accordingly. This way, global evolution is realized through the combination of local evolution and reactivity.

XChange's approach to global evolution requires that a Web site changing its data notifies all dependent Web sites and that appropriate reaction rules are installed locally at the dependent sites. In the current framework of XChange, these reaction rules need to be written manually. How such rules could be derived automatically from some higher-level specifications is an important but different question that is currently outside of the scope of XChange. Once basic reaction and evolution capabilities have been realized with XChange, automatic rule generation might be an interesting research issue.

Chapter 3

An Overview of the Language XChange

An XChange program consists of a number of Event-Condition-Action (ECA) rules together with a number of deduction rules. The program runs locally at a Web site, processes incoming events, can communicate with programs at other Web sites (or even itself!) through sending event messages, and modify persistent data hosted in local or remote Web resources.

XChange embeds Xcerpt, a query language for XML data (and also other semi-structured data), to query volatile data in events and persistent data in Web resources. Xcerpt goes beyond the more traditional select-transform querying known from conventional query languages such as Structured Query Language (SQL) [AHV95, chapter 7.1], XQuery [BCF+03], Lorel [AQM+97], or UnQL [BFS00], and allows reasoning with data by means of deduction rules much similar to logic programming. This also makes Xcerpt an ideal basis for Semantic Web applications.

This chapter gives an overview of Xcerpt and XChange. We first introduce Xcerpt (Section 3.1). Then we turn to XChange and give an overview of the syntax and semantics of its ECA rules (Section 3.2). The three parts of the rules are then analyzed subsequently: the event-part (Section 3.3), the condition-part (Section 3.4), and the action-part (Section 3.5).

3.1 Xcerpt: A Web Query Language

Xcerpt [Sch04] is a declarative, rule-based query language for semi-structured data. It is quite general purpose and can query both tree-structured and graph-structured data. It also allows reasoning with data similar to the logic programming languages. While originally developed for XML, extensions to query RDF [MM04a] data (together with RDF Schema [BG04] data) are underway [FBB05]; investigating extensions to query OWL, Topic Maps, etc., with Xcerpt are interesting research perspectives [BFB+05]. While specialized query languages for each of these formats exist or are being developed, a distinguishing feature of Xcerpt is that one can query many data sources in *different* formats simultaneously. For example, one can query and reason with an (X)HTML document's content and the document's RDF metadata simultaneously.

Figure 3.1 Data term representation of an HTML document

```
html [
    head {
        title { "Example Web Page" },
        meta {
            attributes {
                name    { "author" },
                content { "John Q Public"}
            }
        },
        base  {
            attributes { href {"http://www.example.com"} }
        }
    },

    body [
        h1 [ "An Example Web Page" ],
        p  [ "This is an example Web page." ],
        h1 [ "Another Heading" ]
    ]
]
```

A full introduction to Xcerpt is beyond the scope of this work, but the following should give the reader the general ideas needed to understand XChange and the rest of this work. A tutorial-like, intuitive description of Xcerpt can be found in [SB04]; the full language definition including operational and declarative semantics can be found in the doctoral thesis [Sch04]. A number of research articles have introduced the general underlying ideas [BS02b, BS02a, BS03, BSS04].

3.1.1 Data Terms

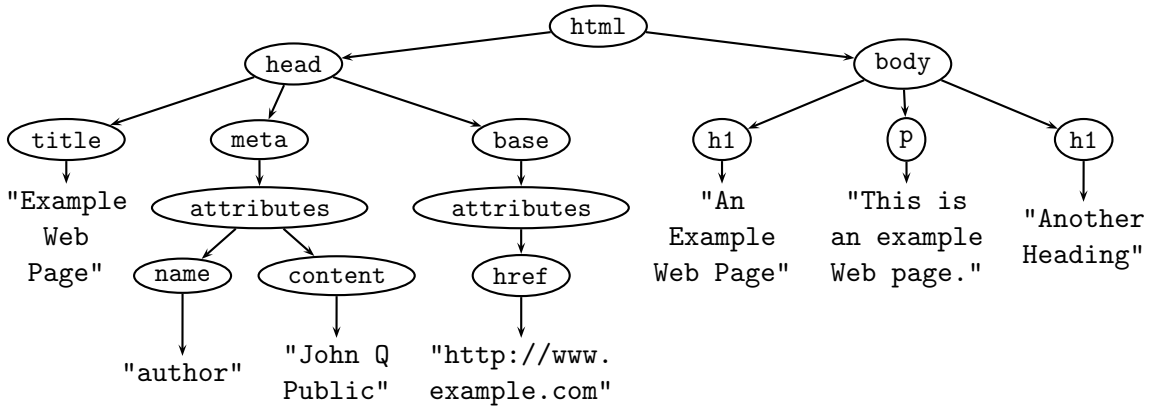
Tree-structured data, e.g., an XML document, is represented by data terms. (Data terms can also represent graph-structured data; we will cover this a little later in this section.) For example, the data term in Figure 3.1 represents the tree in Figure 3.2(b) and the XML document in Figure 3.2(a). Obviously, the XML document is a small XHTML Web page (document type declaration and namespaces ignored). Note that the attributes of an XML-element (e.g., `href="http://www.example.com"` of `<base>`) are collected in a special child with the reserved label `attributes`.

A data term is in essence a pre-order linearization of a tree: the label of the root node is written first, then, inside square brackets `[ ]` or curly braces { }, the linearizations of all its children are written down (separated by commas). Square brackets indicate that the order of a node's children is relevant, curly braces indicate that it is irrelevant. The `attributes`-(sub)term of `meta` in the example of Figure 3.1 is *equivalent* to the term

Figure 3.2 Data terms can represent XML and other tree-structured data

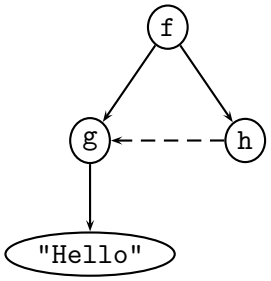
```
<?xml version="1.0" encoding="utf-8"?>

<html>
    <head>
        <title>Example Web Page</title>
        <meta name="author" content="John Q Public">
        <base href="http://www.example.com">
    </head>

    <body>
        <h1>An Example Web Page</h1>

        <p>This is an example Web page.</p>

        <h1>Another Heading</h1>
    </body>
</html>
```

(a) XML Document



(b) Visualization as a tree

Figure 3.3 Graph-structured data. References from the data term are drawn as dotted arrows.




```
attributes {
    content { "John Q Public"},
    name    { "author" }
}
```

(This corresponds to the notion that the attributes of an XML-element form an *unordered* set [CT04, section 2.2].) In contrast, the `body`-(sub)term is *not equivalent* to

```
body [
    p  [ "This is an example Web page." ],
    h1 [ "An Example Web Page" ],
    h1 [ "Another Heading" ]
]
```

Note that data terms are in this respect slightly more general than XML documents: in a XML document the elements are always ordered (so-called document order). But this is not a real limitation of XML since unordered specifications can always be given an order without any loss of information.

Data terms are not limited to tree structure; they can express (rooted) graph structure using a reference mechanism: the construct `id @ t` gives the subterm `t` a name by which it can be referred via `^id`. For example,

```
f {
    r @ g { "Hello" },
    h { ^r }
}
```

represents the rooted graph in Figure 3.3. The reference mechanism `id@t`/`^id` resembles the `ID`/`IDREF`-attributes in XML. Unlike standard XML technologies (e.g., XQuery, XSLT, DOM, SAX) however, Xcerpt makes these references *transparent*. This is, graph edges obtained from references and graph edges obtained from the nested structure of the term are treated the same. For example, the above data term is *equivalent*[1] to

[1]Equivalent here is to be understood the following way: two data terms t_1 and t_2 are equivalent if their respective results of running an Xcerpt query on them are always the same, no matter what the Xcerpt query is.

```
f {
    ^r,
    h { r @ g { "Hello" } }
}
```

At this point, using data term syntax instead of simply the normal XML syntax might seem a little odd to the reader. The term syntax is a little more concise and allows unordered specifications; but this alone does not justify it. The justification will become obvious in the coming sections, when we consider querying data terms. There, conciseness becomes more important for human users, unordered specifications actually play an important role, and we will also need to accommodate other constructs, e.g., partial specifications.

3.1.2 Construct-Query Rules and Goals

An Xcerpt program consists of *construct-query rules*. Rules have the general form

```
CONSTRUCT                        GOAL
    <construct term>                 <construct term>
FROM                      or     FROM
    <query>                          <query>
END                              END
```

where `<query>` selects from existing data and `<construct term>` transforms the selected data (i.e., constructs new data). The following is a very simple Xcerpt program consisting only of one rule:

```
CONSTRUCT
    table-of-contents [
        all entry { var H }
    ]
FROM
    desc var H -> h1 {{ }}
END
```

If run on our example document, the program selects all `h1`-elements (meaning according to HTML: all headings of structuring level 1) and assigns them to the variable H. Then it constructs a new element, `table-of-contents`, and inserts all the headings under it (wrapped in an `entry`-element).

Do not worry about the actual syntax of the `<query>` and `<construct term>`, yet; it will be covered immediately in the next two sections. Observe however that it looks very much like the syntax of data terms. In fact, queries and construction terms describe *patterns* for data terms.

Rules in Xcerpt can be chained: a rule can interact with another rule by querying the "result" of the other rule. This can be compared to views in SQL (a view is created from some tables by a query and can then be queried by other queries just like any table), but rule chaining

is more general and also supports recursion.[2] Rule chaining has a number of applications in a query language [SB04]: code reuse, separation of concerns, mediation between differently structured data sources, and all kinds of recursion (e.g., for a Web crawler or for document transformation via structural recursion as in XSLT [ABC+99]).

Xcerpt rules starting with `CONSTRUCT` do not output the constructed data but only make it available as "input" for other rules via rule chaining. To actually output data, rules starting with `GOAL` instead of `CONSTRUCT` can be used.

In resemblance to logical programming, the query-part of a rule is also called the rule *body* and the construct-part also called *head*.

3.1.3 Query Terms and Queries

A query term represents a *pattern* for data terms. Query terms look very much like data terms; in fact, every data term is also a query term. Additionally, query terms can contain:

- variables (`var X`), which serve as placeholders for arbitrary content;
- variable restrictions (`var X ->` q), which serve as "restricted" placeholders, i.e., the content is not arbitrary but must match the pattern q;
- partial specifications ({{ }}, `[[ ]]`), which omit subterms irrelevant to the query by specifying a pattern (e.g., `a {{ "b", c [[ "d"]] }}`) that matches all data terms (e.g., `a { "b", u["v"], c ["d", "x"], "y"}`) that contain the specified content (in the example, `"b"` and something that matches `c [["d"]]`), but may also contain other content (in the example, `u["v"]`, `"x"`, and `"y"`);
- descendent constructs (`desc` q), which allow to search for a subterm q at *arbitrary depth*;
- and some other constructs, e.g., `optional`, `position`, `without`, which we do not discuss here.

Consider the following example of a query term:

```
html [[
    head {{
        meta {
            attributes {
                name    { "author" },
                content { var A }
            }
        },
        title { var T }
    }}
]]
```

[2] Note that SQL:1999 and later also support recursion (`WITH RECURSIVE ...`).

It describes a pattern that "matches" all data terms where:

1. the root is labeled `html`;
2. there is at least one child of the root, which is labeled `head` (other children may exist);
3. and this `head` node has two children, one labeled `title` and one labeled `meta` (other children may exist, and the order of the children is irrelevant), where
4. the `title` child is not empty, and
5. the `meta` child has exactly two children (order is irrelevant, but no other children may exist), `name` with content `"author"` and `content` with arbitrary content (not empty).

Matching of query term and data term is based on a novel method called *Simulation Unification*. If a query term and a data term successfully *unify* ("match"), it gives as result possible substitutions for the variables occurring in the query term. Simulation Unification of the query term above with our example XHTML document yields exactly one substitution for the variables A and T: $A \mapsto$ `"John Q Public"`, $T \mapsto$ `"Example Web Page"`.

The body of an Xcerpt rule (the part after `FROM`) is a query.[3] A *query* is a connection of query terms with the boolean connectives `and` and `or` (not just binary, but n-ary[4]) and the unary `not`. A query is always associated with one or more resources; these deliver the data terms the query is being evaluated against. A resource can be a document (XML syntax, Xcerpt data term syntax, or other), identified by a URI, an external Xcerpt program, or —by means of rule chaining (see 3.1.2 above)— the program itself. Explicit association with a document happens with the `in`-construct, which contains a `resource`-element giving a URI and input format for the document.

The following is an example of a query used in a rule:

```
CONSTRUCT
    <some construct term>
FROM
    in {
        resource { "http://www.example.org/webpage.xml", "xml" },
        and {
            desc h1 { var H }
        },
        not or {
            desc h2 { var H },
            desc h3 { var H }
        }
    }
END
```

[3] Such a query is also sometimes also called *query proper* to distinguish it from "queries" in SQL, XQuery, etc., which not only select data (query), but also transform it (construct new data).

[4] Another way of saying this is that `and` and `or` have variable arity, i.e., their number of arguments is not fixed. Note that "variable" here means that the arity *varies*, and has nothing to do with variables (e.g., X).

If successful, this query extracts the text of all headings of level 1 (text inside `h1`) that do *not* also appear as headings of level 2 or 3 (text inside `h2` or `h3`) from our XHTML document `http://www.example.org/webpage.xml` and assigns them to the variable H.

A `where`-clause with conditions can be attached to a query in order to further restrict the result with non-pattern constraints. For example,

```
CONSTRUCT
    <some construct term>
FROM
    products {{
        var P -> products {{
            price { var P },
            currency { "EUR" }
        }}
    }}
    where var P < 400
END
```

selects only products with a price less than €400 from some product database.

3.1.4 Construct Terms

The general task of a query language is to *select* and *transform* data [ABS99, chapter 4]. Evaluating the query-part of an Xcerpt rule (cf. previous section) *selects* (or *extracts*) data from some data terms. Now we want to *transform* the selected data; in other words we want to *construct* new data.

More precisely, evaluation of the query part of an Xcerpt rule has delivered a set of variable substitutions (an empty set signifies unsuccessful evaluation). From this substitution set we now want to construct new data terms. *Construct terms*, which are used in the construction part an Xcerpt rule (the part following `CONSTRUCT` or `GOAL`), serve this purpose.

Construct terms are an extension of data terms that may additionally contain variables (*var* X) and grouping constructs (`all` t and `some` n t, where t is a construct term and n an integer). The `out`-construct allows to specify a resource the result should be written to (e.g., a file); its syntax resembles that of `in` (cf. 3.1.3). Note that `out` can only be used in `GOAL`-rules, not in `CONSTRUCT`-rules.

Consider the construct term in the rule in Figure 3.4. The result of the query will be a set of two substitutions, $A \mapsto$ `"John Q Public"`, $T \mapsto$ `"Example Web Page"`, $H \mapsto$ `"An Example Web Page"` and $A \mapsto$ `"John Q Public"`, $T \mapsto$ `"Example Web Page"`, $H \mapsto$ `"Another Heading"`. When constructing a data term from the construct term, variables are replaced by their value from the substitutions. If there is more than one possible substitution and no grouping construct (`all` or `some` n), as is the case here with `var A` and `var T`, one substitution is picked randomly to deliver a value; note that the values of A and T are equal in both substitutions here, so it does not matter which one we pick.

The `all` before `li` introduces a grouping: for each value of H (there are two), a new `li` element is created and `var H` inside it is replaced with its value.

Figure 3.4 Xcerpt rule constructing a table of contents from the XHTML document `http://www.example.com/webpage.xml`

```
GOAL
    out {
        resource { "file:toc.xml", "xml" },
        html [
            head {
                title {"Table of Contents"}
            },
            body [
                h1 [
                    i [ var A ],
                    ": ",
                    b [ var T ]
                ],
                h2 [ "Table of Contents" ],
                ol [
                    all li [var H]
                ]
            ]
        ]
    }
FROM
    in {
        resource { "http://www.example.org/webpage.xml", "xml" },
        html [[
            head {{
                meta {
                    attributes {
                        name    { "author" },
                        content { var A }
                    }
                },
                title { var T }
            }},

            body [[
                desc h1 [var H]
            ]]
        ]]
    }
END
```

If the input XML file is our XHTML document, the rule produces:

```
html [
    head {
        title {"Table of Contents"}
    },
    body [
        h1 [
            i ["John Q Public"],
            ": ",
            b [ "Example Web Page" ]
        ]
        ol [
            li ["An Example Web Page"],
            li ["Another Heading"]
        ]
    ]
]
```

3.1.5 A Visual Query Language based on Xcerpt: visXcerpt

The pattern-based approach used to represent queries and result constructions is very well suited for visualization. A strong visual resemblance between queries and queried data, and result constructions and resulting data allows building a good visual query language that is easy to use even by people with little or no programming experience.

As part of Xcerpt, the visual query language *visXcerpt* has been developed [Ber03, BBSW03]. In visXcerpt, data terms, query terms, and construct terms are represented as nested boxes. For total specifications, the surrounding lines are drawn solid, for partial specifications, dotted. A rule is represented with its construct term on the left hand side and its query on the right hand side, connected with an arrow facing left. Figure 3.5 depicts the visual rendering of the rule from Figure 3.4, which produced a table of contents for an XHTML document. Because rules can become fairly large, visXcerpt's interactive editor allows to hide parts of a pattern by "folding" them. In the example, the content of the `meta`-element on the right hand side is folded in and not shown, and similarly the content of the `i`-element on the left hand side.

3.1.6 Summary

We have seen: Xcerpt programs consist of *rules*. The rule *body* is a query —that is, a boolean formula built from *query terms*— and selects data by finding all possible *substitutions* for variables. The rule *head* is a *construct term* and "transforms" data, i.e., constructs new data from the substitutions obtained in the rule body. Both query terms and construct terms specify *patterns* for *data terms*. Data terms can represent XML documents and other tree-structured or graph-structured data. This pattern-based approach also shows very promising results for visual querying. Rules of an Xcerpt program can be *chained*, which allows reasoning with data through deduction rules.

Figure 3.5 VisXcerpt rendering of the Xcerpt rule in Figure 3.4

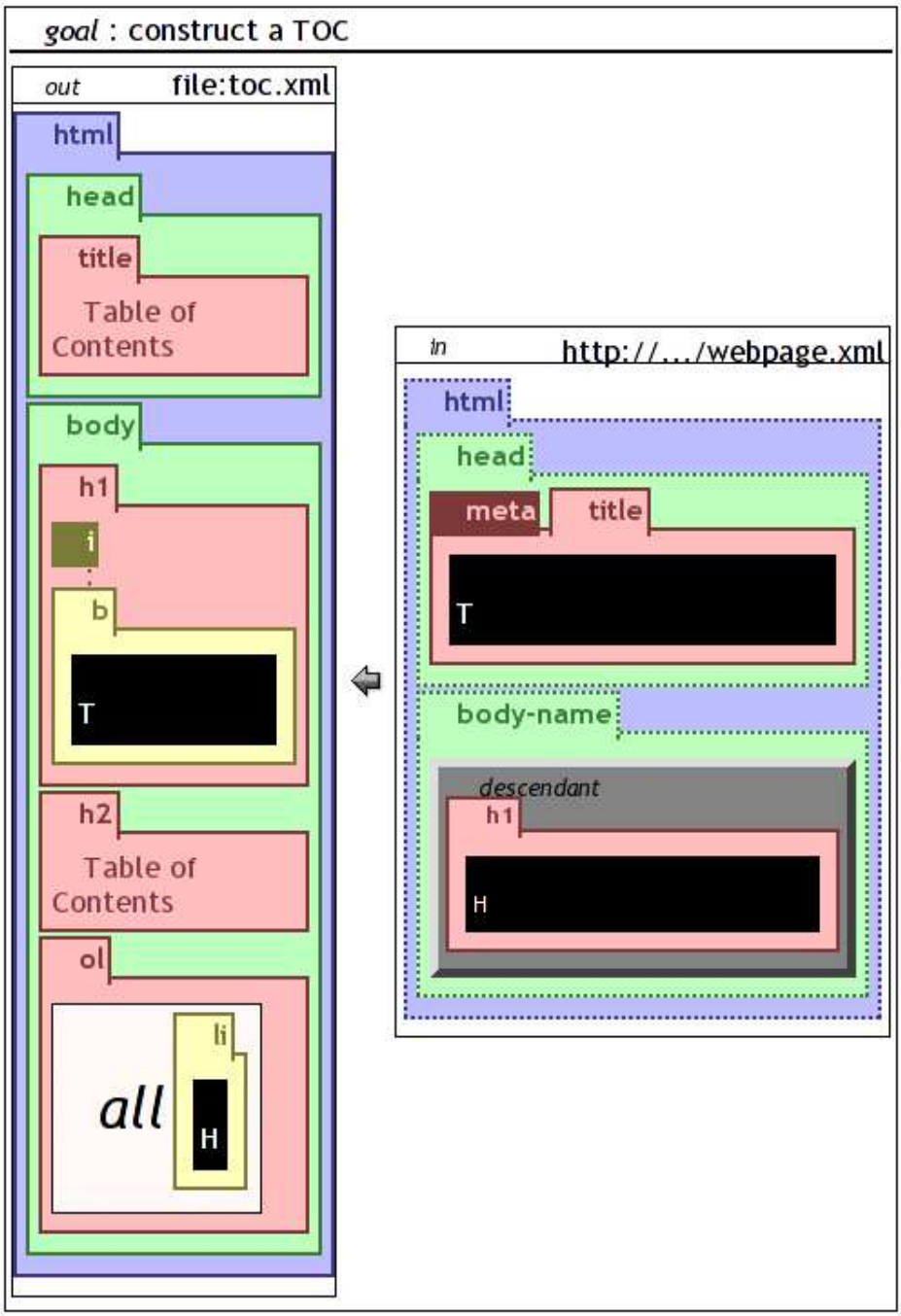


Figure 3.6 Syntax of ECA rules in XChange

```
RAISE
    < event message construction >
ON
    < event query >
FROM
    < Web query (condition) >
END
```

(a) Event raising rule

```
TRANSACTION
    < complex update >
ON
    < event query >
FROM
    < Web query (condition) >
END
```

(b) Transaction rule

3.2 XChange: A Reactive Language for the Web

Query languages like Xcerpt are passive: a user (human or program) has to start a query *manually* and receives a result. Reactivity, in contrast, means that no manual user initiative is necessary. A reactive program reacts *automatically* to certain "events" (changes in the world's state). XChange is a declarative language for programming such reactive behavior in the Web's environment.

A reactive XChange program contains a number of Event-Condition-Action (ECA) rules. An ECA rule specifies to execute a certain *action* as response to an *event*, provided that the *condition* holds. Depending on the type of action, an ECA rule in XChange has one of the two forms depicted in Figure 3.6. Event raising rules (Figure 3.6(a)) state to raise a certain event, i.e., send it to a remote Web site or the local Web site itself. Transaction rules (Figure 3.6(b)) state to execute a complex update in a transactional manner.

An XChange ECA rule reacts on classes of *events*, specified by *atomic* or *composite event queries* in the event-part of the rules (following the keyword `ON`). After the detection of a relevant event, the rule tests its *condition*, which is a *query to (persistent) Web resources* (following the keyword `FROM`). If the test is successful, the *action* (specified after keyword `RAISE` or `TRANSACTION` respectively) is executed. Note that the *effect* of a rule, the action, is written at the top as rule head, while the *cause*, event and condition, is written below as rule body.

As we will see, the three different parts of the rules use a term syntax in the fashion of Xcerpt and can contain variables. Information flows from the event-part to the condition-part and from there to the action-part of a rule via substitutions for the variables.

We will now look closer at the three parts of a rule. Note that, at the time of writing this, XChange is still work in progress; the discussion provided here follows [BFPS04, BP05, BBP04].

3.3 Events in XChange

The event-part of an ECA rule specifies a class of events the rule reacts upon. This class of events is expressed by an *event query*. Event queries access only volatile data, i.e., event data; they cannot access persistent data from Web resources.

The basic building blocks are *atomic events*; they are happenings at a Web site that occur at a single time point, e.g., an update of an element in a local XML file, a timer event, or reception of a message about an event at a remote Web site.

3.3.1 Representation of Atomic Events

Atomic events are represented as XML (we will usually use the compact data term notation introduced above). Because they are often used for communication between different XChange-aware Web sites, we also call the representation of an atomic event *event message*. An envelope with prescribed structure provides management information for atomic events, e.g., the sender of an event or the reception time; apart from these standard fields, structure and content of the event message can be chosen freely to fit an application's needs.

Figure 3.7 shows an event in XML representation (Figure 3.7(a)) and in the equivalent data term representation (Figure 3.7(b)). Obviously, the free content of the event message signifies that John Q Public's flight has been canceled. Details on the envelope structure will be provided later in Chapter 4.

An XChange-aware Web site receives event messages in a stream-like, ordered fashion. We call the sequence of all event messages it receives the *stream of incoming events*, or *event stream* for short.

3.3.2 Purpose of Event Queries

As we can see from this example, an event often contains valuable information that will be needed in the condition-part and action-part of a rule reacting to this particular event, e.g., the flight number or the passenger name. Hence, event queries serve a double purpose:

- they specify classes of events, the rule reacts upon (as we mentioned already), and
- they extract data from events for use in the condition-part and action-part of a rule (in the form of substitutions for variables).

Each time events that allow a successful evaluation of an event query have been received, the condition-part of the corresponding rule is evaluated. We also say that the rule (execution) is *triggered*.

An event query can be an atomic event query or a composite event query. Atomic event queries specify (structural) patterns for event messages to detect single atomic events, while composite event queries specify temporal patterns to detect occurrences of several atomic events over some time.

3.3.3 Atomic Event Queries

Atomic event queries are simply query terms (see 3.1.3); they specify a (structural) pattern for single event messages. An atomic event query is evaluated against each incoming event

Figure 3.7 Representation of an Event

```
<xchange:event xmlns:xchange="http://pms.ifi.lmu.de/xchange">
    <xchange:sender>         http://airline.com   </xchange:sender>
    <xchange:recipient>      http://passenger.com </xchange:recipient>
    <xchange:raising-time>   2005-05-29T18:00     </xchange:raising-time>
    <xchange:reception-time> 2005-05-29T18:01     </xchange:reception-time>
    <xchange:reception-id>   4711                 </xchange:reception-id>

    <flight-cancellation>
        <flight-number> UA917 </flight-number>
        <passenger>
            <name>
                <first>  John   </first>
                <middle> Q      </middle>
                <last>   Public </last>
            </name>
        </passenger>
    </flight-cancellation>
</xchange:event>
```

(a) XML Representation

```
xchange:event [
    xchange:sender         ["http://airline.com"],
    xchange:recipient      ["http://passenger.com"],
    xchange:raising-time   ["2005-05-29T18:00"],
    xchange:reception-time ["2005-05-29T18:01"],
    xchange:reception-id   ["4711"],

    fligh-cancellation {
        flight-number { "UA917" },
        passenger {
            name {
                first  { "John" },
                middle { "Q" },
                last   { "Public" }
            }
        }
    }
]
```

(b) Data Term Representation

Figure 3.8 Atomic event query detecting flight cancellations for `John Q Public`

```
RAISE
    < action-part >
ON
    xchange:event {{
        flight-cancellation {{
            flight-number { var N },
            passenger {{
                name {
                    first  { "John" },
                    middle { "Q" },
                    last   { "Public" }
                }
            }}
        }}
    }}
FROM
   < condition-part >
END
```

message. If they successfully unify, the atomic event query has an answer and the condition-part and action-part of the rule are executed. The variable substitutions obtained in the unification can be used in the condition-part and action-part.

Figure 3.8 shows an example of a rule with an atomic event query (condition-part and action-part have been left out). It detects all events that signify a flight cancellation for passenger `John Q Public` (e.g., the event in Figure 3.7), and places the flight number in variable `var N` for usage in the condition-part and action-part.

3.3.4 Composite Event Queries

Composite event queries are connections of atomic event queries with *composition operators*. The composition operators detect temporal patterns of atomic events to be found in the *stream of incoming events*. For example, the operator `andthen [` eq_1`,` eq_2`,` eq_3 `]` (where the eq_i are atomic or composite event queries) detects successive occurrences of events in the classes eq_1, eq_2, eq_3. Note that the events do not have to be directly successive, but can be interleaved with other events.

XChange provides a rich set of composition operators for events and also temporal restrictions. Every composite query in XChange needs a temporal restriction to achieve a bounded event life-span (cf. 2.2.4) and avoid increasing demand in event storage requirements. We will discuss composition operators and temporal restrictions in Chapter 4.

Composite events, unlike atomic events, have no direct definition in XChange. Instead, the notion of a composite event is only defined as answer to some composite event query, that is, as a sequence of atomic events that answers the composite event query. The rationale for this becomes obvious when considering negation: there is no way to express "*no* event of type x happens between events a and b" without referring to the class x of events (which is, of course, specified as an event query!).

As presented in this work, XChange event queries do not support event consumption and event instance selection [ZU99]. Event consumption means an event used once in an answer to an event query cannot be reused in another answer to this query. Event instance selection allows selection of one answer to a composite event query if several answers are possible. The amount of complexity they add to a language and its semantics can be very hard to comprehend for users. Also, event consumption is very hard to capture in declarative semantics [BM01]. The design assumption for XChange is that, in the application domains considered for XChange, these are of little relevance and their effect often can be captured by considering the data contained in events. Still, XChange is designed in a way that adding event consumption or instance selection do not contradict the language and could be introduced as new, orthogonal axes for event query specification, if deemed necessary.

3.4 Conditions in XChange

The condition of an XChange ECA rule is an Xcerpt query (see 3.1.3). The query can only access persistent data from Web resources; we hence also call if often *Web query* to distinguish it from event queries. The Web query can make use of the variable substitutions provided by the previous evaluation of the event query, but it cannot access the volatile data from the stream of incoming events in any other way. This satisfies the paradigm of the clear separation of volatile and persistent data argued for in Section 2.2.3.

3.4.1 Web Queries as Conditions

When a rule is *triggered*, the Web query in the condition-part is evaluated. If the evaluation is unsuccessful (i.e., we do not find an answer to the query), rule execution is aborted and the action-part will not be executed; if the evaluation is successful (i.e., we find an answer), it delivers substitutions for the free variables of the Web query, and these can be used in the subsequent execution of action-part.

Continuing the plot of John Q Public's canceled flight, Figure 3.9 shows a condition-part of the rule started in Figure 3.8; an excerpt from the queried flight database is shown in Figure 3.10.

The Web query uses the flight number provided in `var N` by the previous event query, checks that it has an entry in the flight database `http://www.example.com/lufthansa.xml` (a persistent Web resource), and extracts the flight's departure (`var F`) and destination airport (`var T`).

3.4.2 Deductive Rules in XChange Programs

Web queries in the condition-part of XChange rules can facilitate rule chaining (as introduced above with Xcerpt) to reason with data. For this, an XChange program can contain deductive Xcerpt rules (`CONSTRUCT` ... `FROM` ... `END`) in addition to ECA rules.

To illustrate usage and usefulness of the deductive rules, we extend our flight database example. Instead of just the one flight database `http://www.example.com/lufthansa.xml`, we now have an additional second flight database `http://www.example.com/united.xml`. The

Figure 3.9 Web query that checks the existence of a flight with number `var N` in a flight database and extracts departure and destination airport into `var F` and `var T`.

```
RAISE
    < action-part >
ON
    < event query detecting John Q Public's flight cancellation; >
    < it gives a substitution for var N (flight-number) >
FROM
    in {
        resource { "http://www.example.com/lufthansa.xml", "xml" },
        flights {{
            flight {{
                number { var N },
                from   { var F },
                to     { var T }
            }}
        }}
    }
END
```

Figure 3.10 Excerpt of the flight database located at `http://www.example.com/lufthansa.xml`

```
flights {
    flight {
        number { "UA917" },
        from   { "FRA" },
        to     { "SEA" }
    },

    flight {
        number { "LH1111" },
        from   { "FRA" },
        to     { "MUC" }
    },

    flight {
        number { "LH1332" },
        from   { "MUC" },
        to     { "LAX" }
    },

    ...
}
```

Figure 3.11 Excerpt of the flight database located at `http://www.example.com/united.xml`

```
flightdatabase {
    departure {
        flight-number { "UA917" },
        airport { "FRA" }
    },

    departure {
        flight-number { "UA4711" },
        airport { "LAX" }
    },

    ...

    destination {
        flight-number { "UA917" },
        airport { "SEA" }
    },

    ...
}
```

databases store flight information using different schemas. The schema of the first database we have seen already (Figure 3.10): a set of `flight`-elements —triples of flight number, departure airport, and destination airport— collected in the root element `flights`. The second database however provides tuples of flight number and departure airport (`departure`-element), and separately tuples of flight number and destination airport (`destination`-element). Figure 3.11 shows an excerpt of this second database.

Making use of rule chaining, we can reformulate our Web query to access both databases in an integrated manner. In addition to the one ECA rule containing the Web query we now also have two deductive Xcerpt rules, one for each flight database; see Figure 3.12. The two deduction rules can be compared to views as they are known from SQL in relational databases; in fact, logical views are a standard way to integrate information from different databases [Ull97]. The first deduction rule does not apply any major transformation to its input (except for the ordered specification). The second rule however is more complicated; it performs a natural join on the `flight-number`-element in `departure` and `destination` via the variable `var N`.

3.5 Actions in XChange

When both event query and Web query of a rule evaluate successfully, the rule has recognized a situation that requires a reaction and its action-part is executed. We also say the rule *fires*. The actions available are: raising a new event, i.e., sending an event message to a remote Web site or oneself, and starting a transaction that performs a (simple or complex) update, i.e., modifying local or remote data.

Figure 3.12 An XChange program consisting of one ECA rule and two deductive Xcerpt rules to illustrate the use of rule chaining in Web queries

```
RAISE
    < action-part >
ON
    < event query detecting John Q Public's flight cancellation; >
    < it gives a substituion for var N (flight-number) >
FROM
    flight [  number { var N },
              from   { var F },
              to     { var T }  ]
END


CONSTRUCT
    flight [  number { var N },
              from   { var F },
              to     { var T }  ]
FROM
    in { resource { "http://www.example.com/lufthansa.xml", "xml" },
        flights {{
            flight {{  number { var N },
                       from   { var F },
                       to     { var T }  }}
        }}
    }
END


CONSTRUCT
    flight [  number { var N },
              from   { var F },
              to     { var T }  ]
FROM
    in { resource { "http://www.example.com/united.xml", "xml" },
        flightdatabase {{
            departure   {{  flight-number { var N },
                            airport { var F }  }},
            destination {{  flight-number { var N },
                            airport { var T }  }}
        }}
    }
END
```

Figure 3.13 Event raising action to inform John Q Public of his canceled flight.

```
RAISE
    xchange:event [
        xchange:recipient [ "http://sms-gateway.org/us/206-240-1087/" ],

        text-message [
            "Hi, John! ",
            "Your flight ", var N,
            " from ", var F,
            " to ", var T,
            " has been canceled."
        ]
    ]
ON
    < event query detecting John Q Public's flight cancellation; >
    < it gives a substitution for var N (flight-number) >
FROM
    < Web query that checks the existence of flight var N in a database >
    < and extracts the departure airport (var F) and the destination >
    < airport (var T) >
END
```

3.5.1 Raising Events

A rule can raise an event by specifying an event message pattern after the keyword `RAISE`. Figure 3.13 continues our flight cancellation example: the action-part of the rule "forwards" the flight cancellation as a human-readable text message to John Q Public's mobile phone (via a gateway service at `http://sms-gateway.org/us/206-240-1087/`).

The event message pattern is a construct term (see 3.1.4) with the following restrictions:

- its root element must be `event` in the namespace `http://www.pms.ifi.lmu.de/xchange` (assumed to be bound to the prefix `xchange:` here), and its children are given in an *ordered* specification (i.e., only square brackets `[ ]` can follow `event`);
- the first child of the root must be `xchange:recipient` and have one or more valid URIs as its children;
- none of the children of the root elements has the label `xchange:sender`, `xchange:raising-time`, `xchange:reception-time`, or `xchange:reception-id`.

The URIs in `xchange:recipient` identify the addressees of the event message. The other envelope elements of an event message (`xchange:sender`, etc.) are provided automatically by the system, so it is not allowed to specify them.

3.5.2 Transactions

Updates are performed in transactions, indicated by the keyword `TRANSACTION` as start of the rule head. Transactions can consist of simple updates, complex updates, and also event message patterns.[5]

Before diving into updates, a remark: as of writing this, the language to specify updates in XChange is still work in progress. The discussion provided here is based on the preliminary work in [BFPS04, BP05, BBP04]; the discussion is neither complete nor can its compliance to the final version of the update language be assured. However, the basic ideas of the update language are pretty clear, and the discussion concentrates mainly on them.

3.5.3 Elementary Updates: Update Terms

Elementary updates are specified as patterns for the data to be updated, augmented with the desired update operators. Of course, these patterns are based on our term syntax, and we call them *update terms*. Update terms are an extension of query terms (to locate the data to be updated); in addition they can contain primitive *update operations* of the form

- `insert` c,
- `delete` q,
- q `replaceby` c,

(where c is a construct term and q a query term). The update operations can stand anywhere in the update term where there can stand a normal subterm in a query term. An update term can contain more than one update operation. Note however that nesting of update operations is not allowed.

An update term operates on the data as follows: It locates all fragments of the data where its querying subterms match. Then it executes the specified update operations. The update operations have the obvious meanings:

- `insert` c : inserts a new subterm that is constructed from the construct term c,
- `delete` q : deletes all subterms that match with the query term q, and
- q `replaceby` c : replaces all subterms matching q with a new subterm constructed from c.

Figure 3.14 shows an elementary update used in an XChange rule. It deletes John Q Public's reservation for an airport shuttle from the Web resource at `http://shuttle.com/reservation.xml`. The update term is contained in an *in* resource specification that indicates this Web resource.

[5] Exchange of events *in a transactional context*, i.e., all effects an event has caused (by firing new rules) are taken back on a rollback of its transaction, is currently not considered in XChange and left for future work. See also the discussion on the support of transactions in XChange in Section 2.4.4.

Figure 3.14 An elementary update that deletes John Q Public's reservation for the airport shuttle bringing him to the departure airport

```
TRANSACTION
    in {
        resource { "http://shuttle.com/reservation.xml", "xml" }

        reservations {{
            delete shuttle-to-airport {{
                passenger { "John Q Public" },
                airport   { var F },
                flight    { var N }
            }}
        }}
    }
ON
    < event query detecting John Q Public's flight cancellation; >
    < it gives a substitution for var N (flight-number) >
FROM
    < Web query that checks the existence of flight var N in a database >
    < and extracts the departure airport (var F) and the destination >
    < airport (var T) >
END
```

For trees, the semantics of the update operations are clear; when dealing with graph-structured data, however the semantics of `delete` for example are not obvious. Consider performing the update operation

```
f {{
    delete h {{ }}
}}
```

on the graph-data in Figure 3.15(a), which is also visualized in Figure 3.15(d) (the reference `^r` is drawn as a dotted arrow). Simply removing `h` from the graph would result in a dangling reference `^r`. To fix this, two approaches are conceivable: also remove all references to `h` (*greedy deletion*), or leave `h` in the graph and remove only the edge from `f` to `h` (*parsimonious deletion*). So two results for the update above are conceivable. The result of greedy deletion is shown in Figure 3.15(b) (visualized in Figure 3.15(e)), and the result of parsimonious deletion in Figure 3.15(c) (visualized in Figure 3.15(f)).

As long as one operates on XML data, this is no problem as it is only tree-structured anyway. For operating on graph-structured data, XChange update terms currently use the semantics of *greedy deletion*. This seems better fitting for the term syntax. Ways to let the user decide whether she wishes parsimonious or greedy deletion are being considered in the XChange project.

Figure 3.15 Possible semantics for deletion in a graph: greedy vs. parsimonious

```
f {
  g {
    "Hello",
    ^r
  },
  r @ h { "World" }
}
```

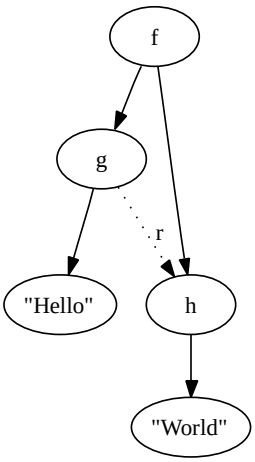
(a) Original term

```
f {
  g {
    "Hello"
  }
}
```

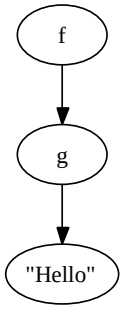
(b) Greedy deletion

```
f {
  g {
    "Hello",
    h { "World" }
  }
}
```

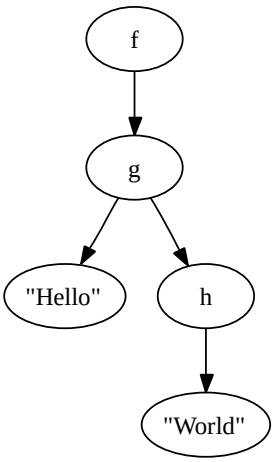
(c) Parsimonious deletion



(d) Original graph



(e) Greedy deletion



(f) Parsimonious deletion

3.5.4 Complex Updates

Elementary updates alone are not very powerful; an elementary update can usually only operate on one single XML document[6] and all data modifications must be captured in one single pattern. To provide a more powerful update mechanism, XChange supports the notion of *complex updates*. Consider the following examples of updates (given in a natural language description) that cannot be realized as one elementary update alone:

- insert a flight reservation at `http://example.com/flights.xml` *and* insert a hotel reservation at `http://example.com/hotels.xml`, *or else* (if either of these updates failed) insert a reservation for the night train at `http://example.com/trains.xml`;
- in a product list with prices in either British pounds or Euro, convert all pounds prices into Euro, *and then* compute the value added tax (VAT) for *all* products and insert a new attribute for it at each product.

A complex update is an ordered or unordered disjunction or conjunction of (simple or complex) updates. A conjunction of updates specifies that all updates have to be successful for the complex update to be successful. If a single update fails, the complex update fails. A disjunction of updates specifies that at least one update has to be successful. The complex update fails only if all updates fail.

Figures 3.16 and 3.17 show complex updates for the two examples above. A complex update is introduced by the keyword `or` for disjunctions or `and` for conjunctions, followed by the constituent updates enclosed in square brackets `[ ]` for an ordered specification or curly braces { } for an unordered specification.

The disjunction and conjunctions can be *ordered* or *unordered*. An ordered specification indicates that the execution order of the updates is relevant. An unordered specification, indicates that execution order is irrelevant, which gives more freedom on parallelization of updates.

Ordered disjunctions implement a *strict or*: After the first update succeeds, execution of the complex update is finished; none of the other updates is executed. Thus, with ordered disjunctions exactly one of the specified updates takes effect. Unordered disjunctions also guarantee that only one of the specified updates takes effect; however, the user expresses no preference which one to try first.

Ordered conjunction of updates is executed as a sequence of the updates, and the result is deterministic, even if the updates access overlapping portions of data. In contrast, the result of unordered conjunctions is nondeterministic if the updates access overlapping portions of data: Consider the product list example from Figure 3.17. If the elementary updates were executed in reversed order, the content of the `vat`-field would have been computed from the price in British pounds for some products in the lower update, but not converted to Euro by the upper elementary update.

[6] One can actually specify more than one XML document in the resource specification `in { ... }`, and thus update several documents at once. But of course this only works if the same modification has to be applied to each document (e.g., we need insert some data record into documents that all have the same schema).

Figure 3.16 Complex update for the "flight and hotel, or train" reservation

```
TRANSACTION
    or [
        and {
            in { resource{"http://example.com/flights.xml", "xml"},
                desc flight {{
                    from { var F },
                    to   { var T },
                    reservations {{
                        insert passenger {
                            name {"John Q Public"}
                        }
                    }}
                }}
            },

            in { resource{"http://example.com/hotels.xml", "xml"},
                desc hotel {{
                    city { var T },
                    reservations {{
                        insert guest {
                            name {"John Q Public"},
                            nights { 1 }
                        }
                    }}
                }}
            }
        },

        in { resource{"http://example.com/trains.xml", "xml"},
            desc train {{
                from { var F },
                to   { var T },
                reservations {{
                    insert passenger {
                        name {"John Q Public"}
                    }
                }}
            }}
        }
    ]
ON
    < obtain substitutions for var F and var T here >
FROM
    < or here >
END
```

Figure 3.17 Complex update of product list: "convert £ to € and then add VAT-field"

```
TRANSACTION
    and [
        in { resource { "http://example.com/products.xml", "xml" },
            desc product {{
                price { var P }
                    replaceby
                    price { (var P * var R) },
                currency { "GBP" }
                    replaceby
                    currency { "EUR" },
            }}
        },

        in { resource { "http://example.com/products.xml", "xml" },
            desc product {{
                price { var Q },
                insert vat { (var Q * var V) }
            }}
        }
    ]
ON
    < does not contain var P >
FROM
    < Obtain exchange rate (var R) and VAT rate (var V) here; >
    < does not contain var P >
```

3.6 Summary

In this chapter, we have given an overview of the reactive language XChange. An XChange program contains *ECA rules*. Additionally it can contain `CONSTRUCT ... FROM ... END` *deduction rules*, which are only relevant in the evaluation of the condition-part of the ECA rules.

An ECA rule reacts on classes of *events*, specified by *atomic* or *composite event queries* in the event-part of the rules. After the detection of a relevant event, a rule tests its *condition*, which is a *query to (persistent) Web resources*. If the test is successful, the *action* is executed. An action can be the *raising of a new event*, or performing a *simple* or *complex update* to Web resources. Information flows from the event-part to the condition part and from there to the action-part in the form of *substitution sets* for the variables.

XChange is built on a pattern-based approach: event queries, Web queries, event raising specifications, and updates are based on terms that resemble the data, which can be represented as data terms. XChange incorporates the Web query language Xcerpt, and extends its notion of query terms and construct terms to update terms.

XChange's ECA rules have a highly declarative syntax and yield a clear separation of volatile and persistent data: volatile data from events can only be queried in the event-part of the

rules, while persistent data from Web resources can only be queried in the condition-part of the rules.

Chapter 4

Event Queries in XChange

XChange provides a rich event query language to detect when a situation of interest (i.e., a situation that might require a reaction) has arisen. Such a situation is, for example, signified by the occurrence of one event or a series of events. Situations that cannot be detected by the occurrence of only one atomic event are called *composite events*. This rather intuitive description of composite events will be refined later.

This chapter introduces XChange's event query language. The syntax will be developed in a stepwise definition of the BNF-grammar. The full grammar is given in appendix A. Along with the query syntax, XChange's semantics of queries will be introduced informally and XChange's notion of answers to event queries will be explained. Formal semantics together with a formal definition of the notion of event query answers are provided in the next chapter.

4.1 Notes on the Terminology

There seems to be no standard terminology for event detection and composite events. Literature uses varying terminology, often different terms for the same concept are used even within one article.

Atomic events are often also called *primitive events* [GJS92a, GD93, CKAK94, BZBW95], and composite events are sometimes also called *complex events* [BZBW95, ZU99] or just *situations* [AE04].

Instead of (composite) event query, the terms *situation definition* [AE04], *(composite) event specification* [GJS92a, CKAK94], *event pattern* [GD93, RC], and *event expression* [ME01, CKAK94] are often used. Likewise, event query languages are often called *situation definition language*, *event specification language*, etc. *Event algebra* is another term used for the event query language as a whole [GD93, BZBW95, BM01] or a certain subset [HV02] of it.

XChange uses the term *event query*, to capture the intuition that one is not only interested in *when* the event takes place, but also in the *data* or *parameters* provided by the events. Also, in this chapter, event query will be sometimes be cut short to query; it will be clear from the context that an event query, not a query to persistent Web data, is meant.

The process of evaluating a (composite) event query is called *(composite) event query evaluation* or *(composite) event detection* synonymously [BM01]. When an event query evaluates

successfully, the event is said to be *detected* or to have *occurred*, and the rule attached to the query is *triggered*.

4.2 Atomic Events

An *atomic event* is a single event that cannot be divided into sub-events and has no duration. It occurs (happens) at a single point in time. In XChange we assume that every relevant atomic event is given a representation in the form of an Xcerpt data term (or XML document) and that this representation contains its occurrence time. We further assume that all atomic events being received by an XChange-aware site, i.e., the *stream of incoming atomic events*, are totally ordered in time; so no two atomic events can happen at the same time. However, the occurrence time representation in an event's data term representation might still be equal for two different events due to limited precision of the representation.

For atomic events, XChange distinguishes between implicit and explicit events.

Explicit events are events that are raised explicitly by rules of an XChange program or by users (e.g., submitting data via a form in an HTML document),[1] and transmitted over the network as messages. For this reason, explicit events —or rather their representation— are also called *event messages*.

In XChange, an explicit event is assumed to occur at the time point when its event message is *received*, not the time it is sent. The rationale is simple: before the event message is received by some node on a network, this node cannot have knowledge that it was raised (sent) by another network node. Attempts to treat raising times of events as occurrence times by assuming (practical) limits about possible network delays and clock synchronization have been made in [MS97]. However, such limits are hard to establish, or maybe even outright unreasonable, on a global network like the Web. Also a delay between reception and detection of a (possibly composite) event is required and leads to losses in performance.

Event messages must have one or more recipients, which can be any XChange-aware Web sites on the network, including the program raising the event itself. Recipients are identified by Uniform Resource Identifiers [BFM98] or (when the standard is adopted) International Resource Identifiers [DS04].

An event message is an XML document or an Xcerpt data term (an abstraction of XML documents, cf. Chapter 3.1.1). It consists of free content inside a fixed-format envelope providing basic information about sender, recipient etc. An exemplary Document Type Definition (DTD) [BPS+04] for the envelope is provided in appendix B, but its format is best understood by looking at an example:

```
<xchange:event xmlns:xchange="http://pms.ifi.lmu.de/xchange">
    <xchange:sender>         http://pms.ifi.lmu.de </xchange:sender>
    <xchange:recipient>      http://example.com    </xchange:sender>
    <xchange:raising-time>   1977-05-20T21:00      </xchange:raising-time>
    <xchange:reception-time> 1978-02-20T12:10      </xchange:reception-time>
    <xchange:reception-id>   4711                  </xchange:reception-id>
```

[1] Note that "user" does not necessarily mean a human user; it can also mean other reactive programs, Web services, Web agents, etc., that are capable of sending XChange-compatible event messages.

```
    <content>
        <more> content </more>
        <even> <more/> content </even>
    </content>
</xchange:event>
```

The corresponding Xcerpt data term representation is:

```
xchange:event [
    xchange:sender         ["http://pms.ifi.lmu.de"],
    xchange:recipient      ["http://example.com"],
    xchange:raising-time   ["1977-05-20T21:00"],
    xchange:reception-time ["1978-02-20T12:10"],
    xchange:reception-id   ["4711"],

    content [
        more [ "content" ],
        even { more {}, "content"}
    ]
]
```

Note that in the XML representation all elements are by definition ordered (in the so-called *document-order*). The Xcerpt data term representation is more general in that it allows both *ordered* specifications, denoted with square brackets `[ ]`, and *unordered* specifications, denoted with curly braces { }. Order specifications signal that the order of sub-terms is significant and must be kept (e.g., the sequence chapters in a book), while unordered specifications signal that the order is insignificant (e.g., database-like set-oriented data). But lack of ordered and unordered specifications is not a real limitation of XML since unordered specifications can always just be put into some (even random) order without loss of information.

As it is apparent from the examples, an event message is a data term (or XML document, respectively) with root label `event` from the XChange namespace (`http://pms.ifi.lmu.de/xchange`), which is assumed to be bound to the prefix `xchange:` here. The root has five or more ordered children; the first five are message information provided by the system, while the rest are arbitrary data terms containing application-dependent information.

Application-dependent information can be any XML data, including XML serializations of data in the Resource Description Framework (RDF) [MM04b] or in the Web Ontology Language (OWL) [MvH04]. This makes XChange apt for Semantic Web applications, and any reasoning capabilities provided by the query language for atomic events (cf. next section) can be used in XChange.

Inside the element `xchange:sender` is the URI of the message *sender*, i.e., the network node raising the explicit event. In the same manner, `xchange:recipient` provides the URI of the *recipient*, i.e., the network node receiving the event message.

The time the explicit event was raised at the sender node is provided in `xchange:raising-time`. It is given according to the local clock of the sender, not the recipient. The time the event message was received at the recipient node is provided in the element `xchange:reception-time`. It is given according to the local clock of the recipient. Note that it is possible for a reception

time provided in an event message to lie *before* the provided raising time because they are given according to two different clocks, which are not necessarily synchronous. Aspects of clock synchronization are beyond the scope of this work. Since we assume that explicit events occur when their event message is *received* such problems are irrelevant in the context of this work.

The element `xchange:reception-id` provides an identifier for each received event message that is locally unique at the recipient node. It enables recipients to distinguish between different event messages with (otherwise) identical content. Thus it renders language constructs like *"event correlation variables"* in [GJS92a] obsolete.

Implicit events, on the other hand, are not raised explicitly, but are caused by some local activity. Implicit events always happen locally. We assume that there is no delay between the rasing time of an implicit event and its "reception" time, i.e., the time the system becomes aware of the implicit event. If an XChange-aware site decides to inform other sites on the network of an implicit event, this has to be done by way of raising an explicit event. Typical implicit events are:

- modifications of (local) persistent data, such as insertion, deletion, or update in an XML document;
- transactional events, such as begin of transaction, commit, or roll-back;
- read operations on (local) data, e.g., the evaluation of some query on an XML document available locally;[2]
- timer events signaling at specified time points;
- system events, such as network failures, system shutdown warnings, or certain (non-lethal) hardware failures (e.g., failure of one disk in a disk array, which does not cause a complete halt of the system).

If an implicit event is relevant to any of the event queries present locally at an XChange-aware site, it is given a standardized representation in the form of an Xcerpt data term. For example a timer event signaling at 12:10 p.m. on February 20, 1978 has the representation:

```
xchange:timer-event [
    xchange:reception-time     [ "1978-20-02T12:10" ]
    xchange:reception-id       [ "42" ],
]
```

By giving implicit events such representations, they can be queried the same way as explicit events. However, some minor restrictions might apply: an event query for a timer event, for example, must specify a *fixed* occurrence time; it is not allowed to use a variable instead, as this would match all the time. Representation of implicit events and limitations on queries to them are still under investigation in XChange; they will thus not be considered specifically in this work.

[2] It might seem unusual that read operations are considered relevant events. But there might be cases when an XChange program has to know about read operations on its local data, e.g., for "manual" concurrency control. Also, read operations can be interesting for adaptive applications, e.g., a data record read very often should be in more prominent locations of a (human-readable) web page.

4.3 Querying Atomic Events

Each rule in an XChange program needs means to specify to which events it reacts (on which events it is triggered). This is done using *event queries* in the event-part of the rules. This section introduces *atomic event queries*, which are used to specify classes of atomic events that require a reaction by some rule and to extract data contained in these events. Atomic event queries serve a double purpose:

1. They specify classes of events to give indication *when* a certain rule is triggered.
2. When a rule is triggered, data contained in the event is needed for the rule execution (condition-part and action-part). Event queries extract data that influences *what* a rule does.

Expressly taking into account the treatment of data in event queries becomes especially important when considering composite events later in this section. In previous work on composite events, this aspect has been rather neglected. Treatment of event data is one of the contributions setting this work apart from previous work on composite events.

With atomic events being represented as semi-structured data, it is only natural to query them like any other semi-structured data, esp. the persistent semi-structured data of Web resources that XChange programs access in the condition-part of rules. Preferably, the query language used for atomic events in the event-part for XChange rules is the same language used for persistent data in the condition-part of XChange rules.

XChange embeds the query language Xcerpt for both, as a query language for atomic events in the event-part and as query language for the condition-part. For events, Xcerpt serves the double purpose called for above:

1. Evaluation of a query against (the representation of) an event can be either successful or unsuccessful. An associated rule is triggered, *when* an event from the class of events with successful evaluation occurs.
2. Successful evaluation of a query gives variable assignments (substitution sets, to be more precise) that are used in the rule execution and influence *what* is done.

In principle, other semi-structured query languages, such as XQuery [BCF$^+$03], XPath [CD99], Lorel [AQM$^+$97], or UnQL [BFS00] could be used as well. However, for the following reasons, Xcerpt seems ideally suited and superior to many of the other available choices:

- Xcerpt has a clean separation of querying —that is, selecting parts of existing data— and construction of the result by transforming the selected data. For event queries in XChange, querying suffices; construction of a result is done in the action-part of XChange rules. And this result should be able to depend on both volatile event data (which was queried in the event-part) and persistent Web data (which was queried in the condition-part).

- Due to this clean separation, Xcerpt provides clear semantics for the variable assignments using the concept of substitution sets. In comparison, XQuery mixes querying and construction and in evaluating a query, so one variable takes different values at different steps of the evaluation (e.g., in different iterations of a `FOR`-loop). The substitution sets of Xcerpt are especially beneficial in an ECA-rule frameworks since they allow to completely evaluate the event-part of a rule before considering the condition-part and action part.

- Xcerpt uses a *positional* or *pattern-based* approach, meaning that a query is like a *form* that gives an *example* of the data that is to be selected. In such an approach, the query pattern closely resembles the queried data. This is easy to grasp for humans and also gives a foundation for visual querying, which has been developed in the project visXcerpt [Ber03, BBSW03].

- The pattern-based approach can not only be used to query data, but also (in a different flavor) to *construct* data or *update data*. It thus gives us a common, unifying framework for the event-, condition-, and action-parts of an ECA rule (cf. Chapter 3).

- The patterns used in Xcerpt do not have to completely specify the structure of the queried data; they can be incomplete in both breadth and depth. That is, certain elements in the queried data can be located without precise knowledge of their position in breadth or depth. This is beneficial if the query programmer is not fully aware of the structure of some event, or if the event's structure evolves.

- Xcerpt naturally supports nested, possibly cyclic, graph data. In contrast, XQuery deals only with tree data. To work with graph-structured data in XQuery, "manual" resolution of references is necessary, e.g., by means of a value join in the `ID/IDREF` attributes.

To query atomic events, only the query (selection) part of Xcerpt is needed, the construction (transformation) part can be ignored in this context. The simplest form of an Xcerpt query is a so-called "query term" and specifies a pattern. This pattern is tried to be matched against an XML document (data term), which will be an event message in this context. The underlying ideas of Xcerpt have already been put forward in Chapter 3.1. Note that we use only query-terms here, the other features of an Xcerpt query (and-, or-connections, resource-specifications) are not needed in the context of atomic event queries.

An atomic event query (symbol `AEQ` in the grammar) is an Xcerpt query term with an optionally attached condition box in the form of a `where`-clause. The queries root label should be `xchange:event` or any of the other prescribed root label for events (e.g., `xchange:timer-event`). The full definition of Xcerpt query terms and condition boxes is provided in [Sch04], and we content ourselves with a comment instead of a full rule for the grammar symbol `AEQ`:

```
AEQ ::= /* Xcerpt Query Term (with optional where clause)*/
```

Shortcut Notation for Atomic Event Queries. Data term representations for events, and hence also atomic event queries, tend to be rather verbose and lengthy. This is rather

inconvenient and unreadable, esp. later in this chapter, when examples for composite events contain more than one atomic event or atomic event query. Therefore we will introduce a shorthand notation and drop the envelope in both event messages and queries. For example, the full event message from above will be shortcut to:

```
content [
    more [ "content" ],
    even { more {}, "content"}
]
```

Keep in mind, however, that this is done only for discussion purposes and not part of XChange's event query language. Also, this shortcut only works if the free content in an event message consists of exactly one element.

4.4 Answers to Atomic Event Queries

An answer to an atomic event query consists of the event message that successfully matched with the query and values for the free variables in the query.

Values for variables are expressed through *substitutions*, which are mappings from all variables in a query to values (data terms). When trying to "match" a query term (atomic event query) and a data term (event message), there might be several substitutions that allow a successful simulation unification. All these substitutions are collected in a *substitution set*. Consider matching the query

```
f {{ var X -> g {{ }}, var Y -> h{{ }} }}
```

and the data term

```
f { g{"1"}, g{"2"}, h{"3"}, h{"4"} }
```

There are four possible substitutions: The first maps X to `g{"1"}` and Y to `h{"3"}`, the second maps X to `g{"1"}` and Y to `h{"4"}`, the third maps X to `g{"2"}` and Y to `h{"3"}`, the fourth maps X to `g{"2"}` and Y to `h{"4"}`. Together this gives a substitution set containing four variables. It is called maximal, because it contains *all* possible substitutions that allow a matching of query term and data term. To just detect events with an event query, we would not really need all substitutions; however, we will need them in the condition-parts and action-parts of our ECA-rules. For example and update term in the action-part could contain `insert i{ all var X }`, and thus need *all* values for X.

To reiterate, an answer to an atomic event query is twofold and consists of

- the event message that matched with the event query, and
- a substitution set, giving all variables substitutions possible in this matching.

4.5 Composite Events

At the beginning of this chapter, composite events were informally introduced as situations of interest that cannot be detected by the occurrence of one single atomic event.

Such a situation of interest (or rather a class of situations) will be defined by a (composite) event query. Accordingly, composite events are sequences of atomic events that *answer a given composite event query*. This means that the notion of composite events in XChange is defined only via the notion of composite event queries (cf. 3.3.4).

Composite events do not occur at a single point in time (as atomic events do), but have a duration. They extend over an interval signified by a *beginning time* and an *ending time*. The ending time, or occurrence time, is also the time the composite event "happens", i.e., the time the rule attached to the composite event's query fires.

4.6 Answers to Composite Event Queries

Before introducing the composite event query language, it is best to look at the notion of answers to composite event queries. Recall that answers to atomic event queries consist of two parts: an atomic event (more precisely, its data term representation, i.e., the event message, together with the event's occurrence time) and a set of substitutions. In this answer, the event message has to "match" the query (the event message's data term and the query term have to unify), and the substitution set gives all assignments for the free variables in the query allowing such a matching.

Answers to composite event queries resemble the combination of event plus substitution set used as answers to atomic event queries: There is one substitution set that gives all assignments for the free variables; this is not different from atomic event query answers. However, composite event queries cannot be answered by one single atomic event. Instead, a number of atomic events must be used to "match" the pattern of events a composite event query specifies. Hence, a composite event query is answered by a sequence of atomic events plus a substitution set.

Answers to atomic and composite event queries are alike in the following way: they consist of

- all atomic events that must have occurred in order to answer the query (which is a single atomic event in the first, and a sequence of —usually more than one— atomic events in the second case) and
- a substitution set giving assignments for the free variables contained in the query.

It might seem puzzling at first that composite event query answers have only one substitution set: if the query is answered by a number of atomic events, why not have a separate substitution set for each of the atomic events? This would conflict with the idea that a variable occurring multiple times in a composite query has to take the same value in all places. Consider the following informal specification of a composite event query from the weather domain, indicating that sightings of rainbows are likely at some place X (where X is a free

variable): "(atomic event) rain in place X, followed by (atomic event) sunshine in place X."[3] The query only makes sense if X has the same value in both places of the query, i.e., if X is a logical variable. Therefore, answers to composite event queries make sense only if just one substitution set is used.

Duration. Composite events stretch over some time interval: they have a beginning and an ending time. Beginning and ending time are not necessarily the occurrence times of the first and last event in the answer sequence. For example, a composite event asking that some event e_1 happens while some other event e_2 does not happen in a given time interval stretches exactly over this time interval. The event e_1 will be part of the answer sequence, and its occurrence time lies somewhere inside the time interval. It does not make sense to reduce the time interval to the occurrence time of e_1, since the whole time interval has to be considered to answer the composite event query (as e_2 is not allowed to happen in this time interval).

Note that any event $e_3 \neq e_1$ happening inside the interval is normally not part of the answer sequence. Though it is needed in evaluating the query (to check that e_2 does not occur, i.e., $e_3 \neq e_2$), the query would still be successful if e_3 had not occurred in the first place. To accommodate cases where *all* events in the interval are needed as part of the answer, XChange event queries support *partial matches* (cf. 4.7.1).

The ending time of the duration interval of a composite event is always the time point at which the rule containing the answered composite event query fires.

Data term representation. For subsequent processing of composite event query answers, the sequence of atomic events is given a representation as one data term (one XML document). The data term representation has the root `xchange:event-seq`. The root's children are all the atomic event messages plus a child indicating the beginning time of the composite event and a child indicating the ending time of the composite event. The children are ordered the following way: the beginning time is the first child, it is followed by the representations of the atomic events in their temporal order, and the ending time is the last child. For example, the data term in Figure 4.1 is a representation of an event sequence that contains two atomic events and lasts from midnight, December 24, 1998 until noon, December 26, 1998.

When using shortcut notation for the atomic events, we will sometimes skip the beginning and ending time of the composite event and write then for example just:

```
xchange:event-seq [
    first-atomic-event { },
    second-atomic-event { }
]
```

Giving event sequences such a data term representation by introducing an "artifical" root, allows to process answers with any usual XML query language.

Note however that, in practice, this data term representation is only of secondary significance: many rules will not need access to the data term representation, since all necessary data from

[3] For this query to make sense, there should be added a constraint on the time difference of the atomic events "rain" and "sunshine". This is possible using the `within`-restriction, which will be introduced later in this chapter.

Figure 4.1 Data term representation of a sequence of atomic events

```
xchange:event-seq [
    xchange:beginning-time [ "1998-12-24T00:00" ],

    xchange:event [
        xchange:sender         ["http://pms.ifi.lmu.de"],
        xchange:recipient      ["http://example.com"],
        xchange:raising-time   ["1998-12-25T21:00"],
        xchange:reception-time ["1998-12-25T21:01"],
        xchange:reception-id   ["42"],

        first-atomic-event { }
    ],

    xchange:event [
        xchange:sender         ["http://xcerpt.org"],
        xchange:recipient      ["http://example.com"],
        xchange:raising-time   ["1998-12-26T11:00"],
        xchange:reception-time ["1998-12-26T11:01"],
        xchange:reception-id   ["78"],

        second-atomic-event { }
    ],

    xchange:ending-time    [ "1998-12-26T12:00" ]
]
```

the event can already be supplied in the variable assignments of the constituent atomic event queries. Given a stream of incoming events, in a reactive language like XChange, one is primarily interested in when and how to *react*. Event queries serve the purpose to *detect* events requiring a reaction and *extracting* data from events. This is in contrast to stream processing [BFO04], where queries *filter* and *transform* the stream.

The Choice of Event Sequences for Answers. Trying to find an event sequence that answers a given query is somewhat similar to searching for a minimal model of a logic formula: in both we look for a minimal set of atomic "items" (events or Herbrand-interpretations for the relations, respectively) that allow successful evaluation of a query/formula. Only, event sequences have a duration and are ordered temporally, while logic models come with (unordered) set semantics.

Using event sequences is, of course, not the only possibility to define a notion of an answer for composite event queries. However, we found that using event sequences keeps things simple, is very intuitive, allows to give a definition of answer that does not require knowledge of the structure of queries, and allows a clear and uncomplicated definition of declarative semantics. A deeper discussion of this issue is provided in Appendix D.

4.7 Composite Event Queries

Composite event queries are built up from atomic event queries using composition operators and temporal restrictions. Composition operators are used to specify combinations of event occurrences such as a conjunction or a sequence of event occurrences. Temporal restrictions are used to limit the time interval in which event occurrences are considered relevant. Additionally, composite event queries can contain variables and condition boxes with restrictions on the variable values.

Temporal restrictions play an important role in bounding the life-span of events, which is necessary to keep storage requirements for events constant (see Section 2.2.4). For this, we will refine our notion of composite event queries later in Section 4.8 to legal event queries, that is, event queries that have an upper bound on the life-spans of all events considered in their evaluation.

XChange's event query language uses a pre- or postfix notation for all language constructs. Since atomic event queries are patterns, they usually stretch over several lines and need proper indentation for readability. Infix composition operators would be very hard to read between the lengthy pattern specifications.

The event query language of XChange provides a rich set of composition operators. This section will now first introduce the three most basic composition operators conjunction, disjunction, and sequence. Next, temporal restrictions are introduced. This is followed by a discussion of the effects of variables in composite events, and an explanation of condition boxes (`where`-clause). This should give the reader a firm understanding of the basic concepts before more composition operators will be introduced.

4.7.1 A First Set of Operators: Conjunction, Disjunction, Sequence

Conjuction A conjunction of event queries specifies that instances of each of the constituent event queries must have been detected to answer the composite query. The necessary events are allowed to occur in arbitrary temporal order. The conjunction in XChange thus resembles the conjunction of Snoop [CKAK94] or Samos [GD93]; it differs from the intersection (written `&`) in COMPOSE/Ode [GJS92b] which requires the constituent events to occur simultaneously.

The keyword `and` indicates a conjuction, and curly braces { } surround the constituent queries to indicate that their temporal order is irrelevant. The `and`-operator has variable arity,[4] but at least one constituent event query has to be present.

```
CEQ ::= "and" "{"  EQ ("," EQ)*  "}"
```

A conjunction of event queries is answered —that is, its corresponding rule is triggered— whenever one of the constituent queries is answered, provided that all other constituent queries have already been answered. Reception of a single atomic event can cause a rule to be fired more than once if different answers (cf. 4.6) containing the received atomic event exist. For example, evaluating the query

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

on the stream

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

of incoming events, will be successful twice upon reception of `b`, one time with answer sequence

```
xchange:event-seq [ a{"1"}, b ]
```

and one time with the answer sequence made

```
xchange:event-seq [ a{"2"}, b ]
```

(using the shortcut notation without beginning time and ending time).

The duration interval of an `and`-composite event covers the duration intervals of all constituent events. That is, the beginning time of the composite event is the minimal (earliest) beginning time of all constituent events (note that these might be composite events themselves), while the ending time is the maximal (latest) ending time of all.

Disjunction An inclusive disjunction of event queries is answered by any of the answers of one of the constituent queries. It is denoted with the keyword `or`. As with `and`, at least one constituent query has to be present, and curly braces { } indicate indifference to their temporal order.

[4] That is, the operator's arity is *not fixed* to binary, but n-ary (with $n > 0$). Note that "variable" here refers to "varying" and has nothing to do with variables in the sense of a variable X.

```
CEQ ::= "or" "{"  EQ ("," EQ)*  "}"
```

XChange also supports exclusive disjunctions which will be introduced later (see 4.7.5).

Sequence Temporally successive occurrences of events can be specified with the sequence operator `andthen`. A sequence (sometimes also called a temporally ordered conjuction) depends on the temporal order of events, and thus square brackets `[ ]` surround the constituent event queries of a sequence. At least two event queries have to be present in the sequence operator.

```
CEQ ::= "andthen" "["  EQ ("," EQ)+  "]"
```

Constituent event queries can be, of course, composite event queries and hence the (composite) event instances can stretch over time intervals rather then simply occur at time points. A sequence of time intervals is understood here the following way: the first time interval has to end before the second time interval starts, and the second time interval has to end before the third starts, etc. "Before" here means strictly before, i.e., $<$ not $\leq$.

Note that for a successful evaluation of an `andthen`-query, the events contributing to an answer do not have to come in direct succession in the stream of incoming atomic events. For example, the query

```
andthen [ a, b ]
```

will successfully evaluate on the stream

```
< a, c, b >
```

of incoming atomic events. The `c` coming in-between does not influence the query evaluation, nor is it part of the answer (which is the sequence containing only `a` and `b`). On the Web, a direct succession of events does not make much sense: since anyone can send a message to a Web site, there is always a possibility that two conceptually successive events are "interleaved" by another event that has nothing to do with these events. This is in contrast to the closed world of active databases where often only events from within the same transaction are considered for some composite event.

In some cases, it is desirable to include in the answer to a query all events that come in-between. Consider an example where the stream of incoming events contains information about stock prices. A user is interested in a (composite) event where a jump of more than 10% of a share listed in the Dow Jones index is followed by a drop of more than 5% of a share listed in the Deutscher Aktien Index (DAX). To react adequately, the user also needs information about changes of other stock prices happening during the composite event's time period. In other words, the answer to the query should include all atomic events that come between the delimiting events of `andthen`. This can be specified using double square brackets `[[ ]]`, which signify so-called "partial matching". For example,

```
andthen [[ a, b]]
```

evaluated on the same stream from above will now include `c` in the answer; so the full answer sequence is

```
xchange:event-seq [ a, c, b ]
```

The grammar rule for `andthen` with partial matching is obvious:

```
CEQ ::= "andthen" "[["  EQ ("," EQ)+  "]]"
```

Example To conclude our discussion of the three basic operators `and`, `or`, and `andthen`, we will now look at a larger example of an event query:

```
andthen [[
    a,
    and { b, c },
    or { e, f }
]]
```

It is being evaluated on the event stream

```
< u, a, v, e, c, w, b, x, e, y, f, f, z >
```

The constituent atomic event query `a` has one answer, the atomic event `a`. The `and`-query has also one answer, the composite event `xchange:event-seq[c, b]`. The `or`-query has four answers, the first and second `e`, and the first and second `f`, though only the last three will be used in composing answers to the whole `andthen`-query.

The query has three answers; their beginning time is always the reception time of `a`, while their ending times are the reception times of the first `e`, the first `f`, and the second `f`. The query is a partial query, so the answer sequences include all atomic events that come between the constituent events. They are:

```
xchange:event-seq [ a, v, e, c, b, x, e ]
xchange:event-seq [ a, v, e, c, b, x, e, y, f]
xchange:event-seq [ a, v, e, c, b, x, e, y, f, f]
```

4.7.2 Temporal Restrictions

Temporal restrictions limit the time interval in which event occurrences are considered relevant. This time interval can be specified absolutely, i.e., by a fixed starting and ending time point, or relatively, i.e., depending on the event occurrence times.

Temporal restrictions are introduced by the keywords `within`, `before`, and `in` following an event query.

Absolute Temporal Restriction An absolute temporal restriction is given by an anchored finite time interval, that is a time interval with a fixed starting and ending time point. The restriction is introduced by the keyword `in` following an event query.

If used on an atomic event query, the restriction allows only atomic events occurring *in* the restriction time interval to answer the query.

If used on a composite event query, the restriction allows only those composite events as answers whose beginning and ending are *in*side the restriction interval. In other words, the restriction forces all constituent atomic events used to answer the composite event query to lie in the specified time interval; atomic events outside this interval are not considered.

```
AEQ ::= AEQ "in" FiniteTimeInterval
CEQ ::= CEQ "in" FiniteTimeInterval
```

It is also possible to leave the starting time point of the restriction interval implicit, i.e., it is given by the time point the query was registered at the XChange-aware site. (Remember that this registration time is always a lower bound for any events considered for an answer, as discussed in Chapter 2.2.4.) The keyword `before` followed by a time point specification is used for this purpose. If it is used on an event query, only those answers are allowed that happen *before* the restriction time point.

```
AEQ ::= AEQ "before" TimePoint
CEQ ::= CEQ "before" TimePoint
```

In XChange, time intervals are denoted as square brackets containing representations of the starting and ending time point, separated by two dots.

```
FiniteTimeInterval ::= "[" TimePoint  ".." TimePoint  "]"
```

In principle, time points can be specified in any string format such as ISO 8601 format [Int00]. The prototype implementation of XChange assumes a specification in the restricted profile of ISO 8601 provided in [WW97], additionally allowing the time zone designator to be dropped to signify the local time of the XChange-aware site the query is running on.

```
TimePoint ::= Year "-" Month "-" Day "T"
              Hour ":" Min (":" Sec ("." Frac)?)? (TZD)?
Year      ::= "0000" | ... | "9999"      /* four digit year */
Month     ::= "01" | ... | "12"          /* two digit month */
Day       ::= "01" | ... | "31"          /* two digit day */
Hour      ::= "00" | ... | "23"          /* two digit hour */
Min       ::= "00" | ... | "59"          /* two digit minute */
Sec       ::= "00" | ... | "59"          /* two digit second */
Frac      ::= [0-9]+
TZD       ::= Z | ("-"|"+") Hour ":" Min
```

For the future development of XChange, it is envisioned to integrate into XChange more sophisticated means of time specifications such as the data formats and calendric data types definable with CaTTS (Calendar and Time Type System) [BRS05].

Relative Temporal Restriction A relative temporal restriction is given by a duration, that is a length of time. The restriction allows only composite events that happen *within* the specified length of time. Their duration, i.e., the difference between ending and beginning time of the composite event, must be less than or equal to the restriction duration in order for them to be answers.

A relative temporal restriction is introduced by the keyword `within` following a composite event query. It cannot be used on atomic events, as those have no duration.

```
CEQ ::= CEQ "within" Duration
```

Durations are specified as (sum of) numbers of years, days, hours, minutes, and seconds. For seconds, positive floating point numbers are allowed, all other units should only be used with positive integers.

```
Duration  ::= (NonNegNum "y")? (NonNegNum "d")?
               (NonNegNum "h")? (NonNegNum "m")?
               (NonNegFloat "s")?
                  /* constraint: string non-empty!*/

NonNegNum   ::= [0-9]+
NonNegFloat ::= [0-9]+   ( "." [0-9]+ )?   ( "e" ("+"|"-")? [0-9]+ )?
```

Just as with absolute temporal restrictions, more sophisticated means to specify durations for relative temporal restrictions are envisioned for a future development of XChange.

4.7.3 Variables

Variables can appear inside the atomic event queries that make up a composite query, but also outside atomic event queries. In the latter case they are bound to (the results of) composite event queries (which can be part of bigger composite query).

Inside Atomic Event Queries Atomic event queries usually contain variables. The same variable can be used in different atomic event queries constituting a composite event query, for example:

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

In general, such variables enforce an equality of their values among all queries they appear in. Evaluation of this example query on the event sequence

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

will fire the corresponding rule once, when `b{1}` is received, with the substitution set containing the one substitution $\{X \mapsto 1\}$. It will not fire upon reception `b{2}`, since $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are "incompatible."

Variables enforcing equality is a top-down perspective on the query: all variable assignments in the (higher-level) composite query must "suit," the (lower-level) constituent queries, i.e., allow successful evaluation of the (lower-level) constituent queries.

Another way of looking at variables is from a bottom-up perspective: if *all* possible variable assignments (i.e., maximal substitution sets) for the (lower-level) constituent queries are given, the variable assignments for the (higher-level) composite query can be computed. The substitution set for the composite query is given by a "join" of the substitution sets for the constituent queries. Join here is to be understood similar to a natural join in relational algebra: given two substitution sets we build their crossproduct and for each tuple of substitutions try to merge them. More formally (where $\perp$ denotes that a value is undefined): $\Sigma_1 \bowtie \Sigma_2 = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \forall X.\ \sigma_1(X) = \sigma_2(X) \vee \sigma_1(X) = \perp \vee \sigma_2(X) = \perp\}$.

For a more complex example involving substitution sets containing more than one substitution, take the following modification of the first example:

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

Here, variable X is part of a partial specification. Consider evaluation of this query on the event sequence

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

Evaluation of the constituent query `a {{var X}}` on `a{1,2}` yields the substitution set $\Sigma_1 = \{\{X \mapsto 1\}, \{X \mapsto 2\}\}$. Evaluation of the constituent query `b {{var X}}` on `b{2,3}` yields the substitution set $\Sigma_2 = \{\{X \mapsto 2\}, \{X \mapsto 3\}\}$. Their join $\Sigma = \{\{X \mapsto 2\}\}$ is non-empty and we have a first successful evaluation of the whole query upon reception of `b{2,3}`, causing execution the corresponding rule. Reception of `b{1,2}` causes another execution, this time with the substitution set is $\Sigma = \{\{X \mapsto 1\}, \{X \mapsto 2\}\}$

Outside Atomic Event Queries Variables can also appear outside atomic event queries, when they are bound to (the result of) some event query with `->` (read "as").

```
AEQ ::= "var" VarName "->" AEQ
CEQ ::= "var" VarName "->" CEQ

VarName ::= [a-zA-Z] [a-zA-Z0-9]*
```

Consider the following example:

```
var X -> and {
    var Y -> andthen [ a {}, b {} ],
    var Z -> c {}
}
```

The variable Z is bound to (the result of) an atomic event query. When the atomic event query evaluates successfully on some incoming event, the variable will be assigned the query answer (cf. 4.4), that is, the incoming event message.

The variables X and Y are bound to (the results of) composite event queries. When the rule is triggered, they are assigned the answers —that is, the data term representation of the sequence of atomic events used in answering the query (cf. 4.6)— of their composite event queries.

If an outside variable is used more than once in an event query, be it either in another outside location or inside an atomic query, the rules about enforcing equality from above apply.

4.7.4 Conditions

Composite event queries can be constrained with conditions on the variables using a where-clause:

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

Condition ::= /* see Chapter 4.5.4 of [Sch04] */
```

The conditions are the same as in a where-clause for atomic event queries and as in Xcerpt; see [Sch04, chapter 4.5.4]. They make it possible to express conditions (such as arithmetic expressions on the variables) that go beyond the structural pattern matching provided by the query terms used in atomic event queries and the temporal pattern matching provided by the composition operators used in composite event queries.

For example,

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

will limit the answers to `and{ a{var X}, b{var Y} }` to those answers where the substitutions satisfy $X < Y$.

4.7.5 More Operators

Exclusions (Negation) Using the `without`-operator, certain event occurrences can be excluded. In other words, one can specify a query that is successful, only if some event does *not* happen. For this, some interval has to be known *during* which said event should not

happen; the interval is specified with the keyword `during` and can be either an absolute time interval, or given by a composite event query.

```
CEQ ::= "without" "{" EQ "}" "during" "{" CEQ "}"
CEQ ::= "without" "{" EQ "}" "during" FiniteTimeInterval
```

An exclusion query `without {EQ} during {CEQ}` is evaluated whenever a composite event query successfully evaluates. An exclusion query `without {EQ} during [tp1 .. tp2]` is evaluated once at the time point `tp2`, that is, when the absolute time interval ends. In both cases, the exclusion query is successful if `EQ` has *failed* to successfully evaluate on the stream of incoming events limited to the time interval provided by `CEQ` or `[tp1 .. tp2]`.

Usage of variables in the event query `EQ` deserves some further explanation. Keeping in mind that variables in general enforce equality, consider

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

being evaluated on

```
< c{2}, c{3}, b[1,2,0], a{1}, d{} >
```

Upon reception of `d{}`, it evaluates successfully once: The composite query `andthen [ c{var Y}, d{} ]` is successful twice, one time with assignment $Y \mapsto 2$ and one time with $Y \mapsto 3$. The query `a{1}` delivers $X \mapsto 1$ for both cases.

Now, with the assignments $X \mapsto 1, Y \mapsto 2$ the query for `b[var X, var Y, var Z]` evaluates on `b[1,2,0]` successfully, causing a failure of the whole query for $X \mapsto 1, Y \mapsto 2$.

But for the assignments $X \mapsto 1, Y \mapsto 3$, evaluation of `b[var X, var Y, var Z]` on `b[1,2,0]` fails, making the whole query successful for $X \mapsto 1, Y \mapsto 3$.

Note that the variable Z cannot be given an assignment when the whole query evaluates with success. It occurs only inside a `without`, and the `without` is successful exactly when Z *cannot* be given a assignment. In other words, Z does not occur in a defining position. Since it has no value, Z may not be used in other parts of XChange-rules, e.g., not in the RAISE-part of a rule. This intuition will be refined later in the next chapter when introducing a distinction between negative (defining) and positive (non-defining) variable occurrences.

The answer sequence for a `without ... during ...`-query will be the answer sequence to the composite event query following `during`, or an empty sequence with beginning time `tp1` and ending time `tp2` if a time interval `[tp1 .. tp2]` follows `during`.

Quantifications Quantifications allow to specify that a certain event happens at least, at most, or exactly a given number of times. In the same manner of the `without`-operator, a limiting interval has to specified for this by means of a composite event query or by giving an absolute time interval. The grammar rules are:

```
CEQ ::= "times" Mult ("any" VarList)? "{" EQ "}" "during" "{" CEQ "}"
CEQ ::= "times" Mult ("any" VarList)? "{" EQ "}" "during" FiniteTimeInterval

Mult    ::= (atleast|atmost)? NatNum
NatNum  ::= [1-9] [0-9]*

VarList a::= var VarName (, var VarName)*
```

where `Mult` specifies the number of times the event should (at least, at most, or exactly) occur. The number must be greater or equal 1. The optional keyword `any` introduces a list of variables (`VarList`) that are existentially quantified; this will be explained shortly after the basics have been covered.

A `times`-query is evaluated whenever the interval specified after the keyword `during` ends. To succeed, the query following `times` has to evaluate successfully at least, at most, or exactly the specified number of times in the interval. As always, variables in this query enforce equality, so only evaluations delivering the same variable assignments count. The individual answer sequences must differ in at least one atomic event instance, however. Consider the query

```
times 2 {
    b{X}
} during {
    andthen[
        a{},
        c{}
    ]
}
```

being evaluated on

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

When `c{}` is received, it evaluates successfully once for $X \mapsto 1$. Evaluation fails for $X \mapsto 2$ since `b{2}` is received too often, and for $X \mapsto 3$ since `b{3}` is received not often enough.

This equality constraint on variables can sometimes be too constraining. Consider an example of e-mail reception: a user wants to be informed (e.g., sent a message to his cell phone) when he receives three e-mails from the same person. The informing message should not only contain the name of the sender, but also the subject lines of the e-mails. The subject lines can be different, so the query

```
times 3 {
    e-mail {
        from    [ var F ],
        subject [ var S ],
        content [[ ]]
    }
} during [tp1 .. tp2]
```

does not meet the user requirements. However, simply dropping the variable S by replacing `subject [ var S ]` with `subject [[ ]]` is not an option, as some "handle" on the subject lines is needed in the action-part of the corresponding rule.

To achieve this aim, the keyword `any` allows to introduce variables that are existentially quantified, meaning that they can have different assignments. The query

```
times 3 any var S {
    e-mail {
        from    [ var F ],
        subject [ var S ],
        content [[ ]]
    }
} during [tp1 .. tp2]
```

matches the user requirements. When the query is successfully evaluated, a set of three substitutions is delivered that agree on F but not necessarily on S.

To give a further example, a query like

```
times 4 { var X -> EQ } in { CEQ }
```

does not make sense; it can never be satisfied because four *different* answers to `EQ` are sought for by the `times`-operator while `var X ->` tries to enforce their *equality*. To make sense out of this query, the variable X should be existentially quantified with `any`:

```
times 4 any var X { var X -> EQ } during { CEQ }
```

The answer sequence to a `times ... during ...` query includes all atomic events needed to answer the event query after `times` and also —if we use a composite event query, not a time interval after during— all atomic events needed answer this composite event query.

Multiple Inclusions and Exclusions Multiple inclusions and exclusions allow to detect situations where out of n specified event queries at least, at most, or exactly m (where $1 \leq m \leq n$) evaluate successfully in some limiting interval. Again, the limiting time interval can be specified by means of a composite event query or by giving an absolute time interval. And again, the whole query is evaluated when the composite event evaluates successfully, or respectively when the specified time interval ends.

The grammar rules for multiple inclusions and exclusions are:

```
CEQ ::= Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                                "during" "{" CEQ "}"
CEQ ::= Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                                "during" FiniteTimeInterval
```

The symbol `Mult` (introduced above with the `times`-operator) sets the multiplicity in the form `atleast` m, `atmost` m, or just m.

If one of the n constituent event queries has more than one answer, this leads to multiple answers for the composite query.

Note that `1 of` $\{EQ_1, \ldots EQ_n\}$ `during` $\{CEQ\}$ provides the exclusive disjunction promised above (see 4.7.1). Although it might seem possible to write `times` m $\{EQ\}$ `during` $\{CEQ\}$ as m `of` $\{ EQ_1, \ldots EQ_m \}$ `during` $\{CEQ\}$ where $EQ_1 = \ldots = EQ_m = EQ$, this is not the same! A single answer to EQ will answer all $EQ_1, \ldots EQ_m$ and thus cause successful evaluation of m `of` $\{ EQ_1, \ldots EQ_m \}$ `during` $\{CEQ\}$.

Variables enforce equality as they always do. All things said about variables in discussion of the `times`-operator transfer in the obvious way to the `of`-operator, including the use of the optional `any`.

4.8 Legal Event Queries

Evaluation of event queries as they have been presented up to now can need an unbounded life-span for events. As argued in Chapter 2.2.4, this blurs separation of volatile and persistent data and can lead to monotonically increasing event storage requirements.

We now restrict our notion of event queries to so-called *legal event queries* that can be evaluated with events of bounded life-span only. Chapter 6 gives a proof of this claim based on the declarative semantics defined in Chapter 5.

The following event queries are legal:

- atomic event queries (including temporally restricted atomic event queries),
- composite event queries temporally restricted to a finite absolute time interval with `in`,
- composite event queries temporally restricted to happen before some point in time with `before`,
- composite event queries temporally restricted to a finite duration (finite relative time interval) with `within`,
- exclusions (`without`), quantifications (`times`), multiple inclusions and exclusions (m `of`) where the limiting time interval (following the keyword `during`) is given as an absolute time interval.

As grammar rules we can write:

```
LEQ ::= AEQ
      | CEQ "in" FiniteTimeInterval
      | CEQ "before" TimePoint
      | CEQ "within" Duration
      | "without" "{" EQ "}" "during" FiniteTimeInterval
      | "times" Mult ("any" VarList)? "{" EQ "}" "during" FiniteTimeInterval
      | Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                              "during" FiniteTimeInterval
```

The definition of legal event queries we have chosen here strives for simplicity and clarity; it should be easy to remember for users of the language. It gives sufficient, but not necessary conditions for event queries that can be evaluated with events of bounded life-span only. That is, there are event queries that can be evaluated with bounded event life-spans, but are not considered legal by this definition. However, most of the time, it should be possible to reformulate such an event query as an equivalent legal event query. To give an example, the event query

```
and {
  x {} in [b1 .. e1]
  y {} in [b2 .. e2]
}
```

is not legal by our definition, but equivalent to the legal event query

```
and {
  x {} in [b1 .. e1]
  y {} in [b2 .. e2]
} in [b .. e]
```

where the time point `b` is the minimum of `b1` and `b2` and `e` the maximum of `e1` and `e2`.

Chapter 5

Declarative Semantics for Event Queries

Having discussed the language constructs for event queries informally and intuitively in the previous chapter, we now provide formal, declarative semantics. Formal semantics are desirable for several reasons [Sta95, 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. An concrete example here is the need for duplicate elimination (cf. Chapter 7.3.6) in the incremental event query evaluation we will describe in a later chapter; without work on formal semantics, duplicate elimination it 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. In Chapter 6, we will use the formal semantics for XChange event queries defined here to prove that for every legal event query, there is some upper bound on the life-spans of all events needed to evaluate.

- **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 $\lhd$ 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 the introduction of some notation. It will be introduced as we proceed and is summarized in Appendix C.

5.1 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 [Sch04]. For the understanding of this chapter, only little knowledge about data terms, simulation unification, etc., is required. We avoid giving 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 [GJS93, CKAK94], 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".[1] 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. 2.2.4).

The stream of incoming events $\mathcal{E}$ is a finite sequence $\langle a_1, a_2, \ldots a_n \rangle_b^e$ 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: $b \leq r_1 < \ldots < r_n \leq e$. The total ordering of atomic events corresponds to the assumption that no two atomic events happen simultaneously (cf. 4.2). 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 $a = d^r$.

5.2 Answers to Event Queries

Answers. The notion of answers to event queries 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.

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 $\sigma = \{X \mapsto f["42"], Y \mapsto "abc"\}$ to indicate that σ assigns the term $f["42"]$ to the variable named X, "abc" to Y, and no value to all other variables.[2]

[1] 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 7

[2] This is a simplified definition. In [Sch04], substitutions map *all* variables to *construct* terms (where $X \mapsto X$ corresponds to no assignment in our simplified definition). These, the slightly more complicated definition is

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

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

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: $b \leq rcp(a_1) < rcp(a_2) < \ldots < rcp(a_n) \leq e$

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 $\subset$. The requirement is then $s \subset \mathcal{E}$. However, later in this chapter the right hand side can also be an arbitrary event sequence s' (other then $\mathcal{E}$), giving $s \subset s'$. Formally we define:

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

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

An event sequence can also be a complete subsequence (a substring) of the event stream, that is, it contains *all* atomic events from the event stream that lie between the sequence's

necessary to define semantics of Xcerpt rules which include a construction part (cf. 3.1). The definition of 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 [Sch04]

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 ($s \sqsubset s'$):

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

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

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

Union of event sequences. For later use when defining the answer relation $\lhd$, 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:

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

- $\{a''_1, \ldots a''_p\} = \{a_1, \ldots a_n\} \cup \{a'_1, \ldots a'_m\}$, and
- $b'' \leq a''_1 < \ldots < a''_p \leq e''$, 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 $s_1 \cup \ldots \cup s_n$.

5.3 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 of incoming atomic event $\mathcal{E}$. We write $q \lhd_{\mathcal{E}} s, \Sigma$ to express that query q is *answered by* answer (s, Σ) under the event stream $\mathcal{E}$. Recall that we allowed dropping of the parentheses so that $q \lhd_{\mathcal{E}} s, \Sigma$ is just short form of $q \lhd_{\mathcal{E}} (s, \Sigma)$.

It deserves some justification why we use this special answering relation $\lhd_{\mathcal{E}}$ 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 no 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 [Sch04] 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 $\Phi\ |_U \subseteq \Sigma\ |_U$. (See also [Sch04].) 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.

5.3.1 Atomic Query

Case $q \in \mathcal{T}^q$, i.e., q is an atomic event query ($\mathcal{T}^q$ denotes the set of all possible Xcerpt query terms).

$q \lhd_{\mathcal{E}} s, \Sigma$ 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 [Sch04, 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 substitution 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*[3] resulting from replacing all variable occurrences in q by their value according to σ (Note that for every variables 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 [Sch04].

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.

5.3.2 Conjuction

Case $q = \texttt{and}\{q_1, \ldots\ q_n\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ 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`.

5.3.3 Disjunction

Case $q = \texttt{or}\{q_1, \ldots\ q_n\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

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

- $q_i \lhd_{\mathcal{E}} s, \Sigma$ 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.

5.3.4 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 $end(s) = end(s_2)$.

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 signify 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 $\texttt{andthen}[[q_1, q_2, q_3, \ldots q_n]] \mapsto \texttt{andthen}[[q_1, \texttt{andthen}[[q_2, q_3, \ldots q_n]]]]$.

5.3.5 Absolute Temporal Restriction

Case $q = q'\ \texttt{in}\ [b..e]$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

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

Case $q = q'\ \texttt{before}\ e$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

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

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.

5.3.6 Relative Temporal Restriction

Case $q = q'\ \texttt{within}\ w$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

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

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

5.3.7 Variable Restriction

Case $q = \texttt{var}\ X \rightarrow q'$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

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

5.3.8 Exclusions

Case $q = \texttt{without}\ \{q_1\}\ \texttt{during}\ \{q_2\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if

- $q_2 \lhd_{\mathcal{E}} s, \Sigma$
- For all $s' \subset \mathcal{E}$ 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 $q = \texttt{without}\ \{q_1\}\ \texttt{during}\ [b..e]$.

Variation on the case above: remove $q_2 \lhd_{\mathcal{E}} s, \Sigma$ and replace $begin(s')$ with b and $end(s')$ with e.

5.3.9 Quantifications

Case $q = \texttt{times}\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{q'\}\ \texttt{during}\ \{q''\}$.

(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 $s' \subset \mathcal{E}$ 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 $q =$ `times atleast` n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $\{q''\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ 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

Case $q =$ `times atmost` n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $\{q''\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ 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

Cases $q =$ `times` (`atleast`|`atmost`)? n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $[b..e]$.

Variations on the above cases: replace $q'' \lhd_{\mathcal{E}} s'', \Sigma$ 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.)

5.3.10 Multiple Inclusions and Exclusions

Case $q = m$ `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $\{q''\}$.

(Note that the keyword `any` is dropped in the case $k = 0$ and that $m \geq 1$.)

$q \lhd_{\mathcal{E}} s, \Sigma$ 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 $\iota : \{1, \ldots m\} \rightarrow \{1, \ldots n\}$, and an event sequence s'' such that

- $s = s'' \cup \bigcup_{1 \leq i \leq m} s_i$
- $q'' \lhd_{\mathcal{E}} s'', \Sigma$
- $q_{\iota(i)} \lhd_{\mathcal{E}} s_i, \Sigma_i$ 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 $q_1 \lhd_{\mathcal{E}} s_i, \Sigma_i$) for all $1 \leq i \leq n$
- if there exists an $s' \subset \mathcal{E}$ 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 $q_j \lhd_{\mathcal{E}} s', \Sigma'$, 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 the be answered; the injection ι tells which. The third line of the definition demands that there are at least m answered query, the last line that there are no more than m. Existential quantification of variables with `any` is taken care of in line 5.

Case $q = \texttt{atleast}\ m\ \texttt{of any var}\ X_1, \ldots \texttt{var}\ X_k\ \{q_1, \ldots q_n\}\ \texttt{during}\ \{q''\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $p \geq mm$ 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

Case $q = \texttt{atmost}\ m\ \texttt{of any var}\ X_1, \ldots \texttt{var}\ X_k\ \{q_1, \ldots q_n\}\ \texttt{during}\ \{q''\}$.

$q \lhd_{\mathcal{E}} s, \Sigma$ holds if and only if there exist $1 \leq p \leq mm$ 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

Cases $q = (\texttt{atleast}|\texttt{atmost})?\ m\ \texttt{of any var}\ X_1, \ldots \texttt{var}\ X_k\ \{q_1, \ldots q_n\}\ \ \texttt{during}\ [b..e]$.

Variations on the above cases: replace $q'' \lhd_{\mathcal{E}} s'', \Sigma$ with $s'' = \langle\rangle_b^e$.

Chapter 6

Legal Event Queries and Bounded Event Life-Span

Based on the declarative semantics from the previous chapter, we will now prove that every legal event query needs only events of bounded life-span to be evaluated, as promised in Chapter 2.2.4.

We start by recalling the definition of legal event queries (Section 6.1). Then we formulate what we want to prove as a theorem (Section 6.2), which will subsequently be proven by two lemmas (Sections 6.3 and 6.4). We conclude the chapter with a short discussion of the proof and indications where it has to modified or extended if the definition of legal event queries is changed or new language constructs are introduced in XChange. (Section 6.5).

6.1 Reminder: Definition of Legal Event Queries

In Chapter 4.8 we defined legal event queries. As a reminder, we reiterate this definition here but follow the notation used in defining the declarative semantics (Chapter 5).

Let LEQ be the set of a legal event queries. We then have $q \in \mathsf{LEQ}$ if and only if:

- $q \in \mathcal{T}^q$, i.e., q is an atomic event query; or
- $q = q'$ `in` $[b'..e']$; or
- $q = q'$ `before` e'; or
- $q = q'$ `within` w'; or
- $q =$ `without` $\{q_1\}$ `during` $[b..e]$; or
- $q =$ `times` (`atleast`|`atmost`)? n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $[b..e]$; or
- $q =$ (`atleast`|`atmost`)? m `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $[b..e]$.

Analogously let EQ denote the set of all event queries; a definition in the style above has already been given by the case distinctions in Chapter 5.

6.2 Theorem: Evaluation with Bounded Life-Span

An event query is being evaluated against a stream of incoming events. For each query, this stream starts only when the rule containing the query is registered; an event query "sees" only events happening after its registration and cannot look into the "past" (cf. Chapter 5.1 and also Chapter 2.2.4).

Therefore we can consider each legal event query individually, with its individual stream of incoming events, when proving that it can be evaluated with events of bounded life-span only.

Remember that we defined the stream of incoming events $\mathcal{E}$ as a finite sequence of atomic events $\langle a_1, a_2, \ldots a_n \rangle_b^e$ occurring between time point b and e. Here, b is the time that the event query was registered; since we consider each event query individually, we can assume $b = 0$ for simplicity.

So, given some legal event query $q \in \mathsf{LEQ}$ we want to show that there exists some fixed bound $\beta \in \mathbb{D}$ (a relative time interval or, in other words, a length of time) such that at any point of time $t \in \mathbb{T}$ in order to evaluate q we only need an extract (or restriction) of the stream of incoming events $\mathcal{E}$ whose length is bounded by β. The bound β is determined only from the query q and is independent of the event stream $\mathcal{E}$.

We will denote a restriction (an extract) of the event stream $\mathcal{E} = \langle a_1, a_2, \ldots a_n \rangle_0^t$ that starts at time point b and ends at time point e with $\mathcal{E} \mid_b^e = \langle a_i, a_{i+1}, \ldots a_j \rangle_{b'}^{e'}$ where $b' \leq rcpt(a_i) \leq rcpt(a_{i+1}) \leq \ldots \leq rcpt(a_j) \leq e$, i minimal, j maximal, and $b' = \min\{0, b\}$ and $e' = \max\{e, end(\mathcal{E})\}$.

This gives us the following theorem to prove:

Theorem. For all legal event queries $q \in \mathsf{LEQ}$, there exists a time bound $\beta \in \mathbb{D}$ (a length of time), such that for all time points $t \in \mathbb{T}$, all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq t$), and all answers (s, Σ) with $end(s) = t$ we have:

$$q \lhd_{\mathcal{E}} s, \Sigma \quad \Longleftrightarrow \quad q \lhd_{\mathcal{E}|_{t-\beta}^{t}} s, \Sigma.$$

Proof. The theorem follows immediately from the two lemmas below; for some given legal event query q, a bound β is determined by lemma 1.

6.3 Lemma 1: Bound for the Answers to Legal Event Queries

We first prove that, for a given legal query, the duration of all its answer sequences is bounded by some fixed bound β.

Lemma 1. For all legal event queries $q \in \mathsf{LEQ}$, there exists a time bound $\beta \in \mathbb{D}$ (a length of time), such that for all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq t$), and all answers (s, Σ) with $end(s) = t$ we have:

$$q \lhd_{\mathcal{E}} s, \Sigma \quad \Longrightarrow \quad end(s) - begin(s) \leq \beta.$$

Proof. By case distinction on the definition of LEQ; the definition is not inductive, so the proof is not either. Note during the proof that $begin(s)$ and $end(s)$ are always finite time points; they never become (positive or negative) infinity.

Case $q \in \mathcal{T}^q$, i.e., q is an atomic event query.
Let $\beta = 0$. The definition of $\lhd_{\mathcal{E}}$ for q atomic event query gives us $s = \langle d^r \rangle_r^r$, and we have $end(s) - begin(s) = r - r = 0 \leq \beta$.

Case $q = q'$ `in` $[b..e]$.
Let $\beta = e - b$. From $b \leq begin(s)$ and $end(s) \leq e$ (cf. definition of $\lhd_{\mathcal{E}}$ for this case) we conclude $end(s) - begin(s) \leq e - b = \beta$.

Case $q = q'$ `before` e.
Let $\beta = e - 0$. From $begin(\mathcal{E}) \leq begin(s)$ and $end(s) \leq e$ we conclude (remember that we assumed $begin(\mathcal{E}) = 0$): $end(s) - begin(s) \leq e - begin(\mathcal{E}) = e - 0 = \beta$.

Case $q = q'$ `within` w.
Let $\beta = w$. Immediately from the definition we get $end(s) - begin(s) \leq w = \beta$.

Case $q =$ `without` $\{q_1\}$ `during` $[b..e]$.
Let $\beta = e - b$. As in the `in`-case, definition of $\lhd_{\mathcal{E}}$ gives us $b \leq begin(s)$ and $end(s) \leq e$, and so $end(s) - begin(s) \leq e - b = \beta$.

Case $q =$ `times` (`atleast`|`atmost`)? n `any var` $X_1, \ldots,$ `var` X_k $\{q'\}$ `during` $[b..e]$.
Let $\beta = e - b$. From $s = s'' \cup \bigcup_{1 \leq i \leq n} s_i$ we get $begin(s) = \min\{s'', s_1, \ldots s_n\}$ and $end(s) = \max\{s'', s_1, \ldots s_n\}$. With $begin(s'') \leq begin(s_i)$ and $end(s_i) \leq end(s'')$ for all $1 \leq i \leq n$ and $s'' = \langle \rangle_b^e$ (i.e., $begin(s) = b$ and $end(s) = e$) we are done: $end(s) - begin(s) \leq end(s'') - begin(s'') = e - b = \beta$.

Case $q =$ (`atleast`|`atmost`)? m `of any var` $X_1, \ldots$ `var` X_k $\{q_1, \ldots q_n\}$ `during` $[b..e]$.
Analogously to the `times`-case: let $\beta = e - b$ and use $s = s'' \cup \bigcup_{1 \leq i \leq m} s_i$, $begin(s'') \leq begin(s_i)$, $end(s_i) \leq end(s'')$, and $s'' = \langle \rangle_b^e$.

6.4 Lemma 2: Restriction of the Event Stream

With the previous lemma we have, for each legal event query, established a bound on the duration of answers. We will now prove that for finding some answer (s, Σ), the evaluation of the query has only to consider those events from the event stream $\mathcal{E}$ that lie between the beginning time $begin(s)$ and ending time $end(s)$ of the answer. We prove this for arbitrary event queries, not just legal event queries. With this and Lemma 1, the theorem from above immediately follows.

Lemma 2. For all event queries $q \in \mathsf{EQ}$, all event streams $\mathcal{E}$, and all answers (s, Σ) we have:

$$q \lhd \mathcal{E}\ s, \Sigma \quad \Longleftrightarrow \quad q \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}}\ s, \Sigma.$$

Auxiliary Proposition. Before starting the proof of Lemma 2, we formulate an auxiliary proposition for our subsequence relations $\sqsubset$ and $\subset$:

$$\forall b \leq begin(s) \forall e \geq end(s). \quad s \sqsubset \mathcal{E}\ |_{begin(s)}^{end(s)} \iff s \sqsubset \mathcal{E}\ |_b^e$$

and

$$\forall b \leq begin(s) \forall e \geq end(s). \quad s \subset \mathcal{E}\ |_{begin(s)}^{end(s)} \iff s \subset \mathcal{E}\ |_b^e$$

From left to right we use that $\mathcal{E}\ |_{begin(s)}^{end(s)} \sqsubset \mathcal{E}\ |_b^e$ by definition of the restricted event stream and the transitivity of $\sqsubset$. From right to left we show that all atomic events a that are in the sequence s and in the restricted event stream $\mathcal{E}\ |_b^e$ are also in $\mathcal{E}\ |_{begin(s)}^{end(s)}$. This follows directly from $begin(s) \leq rcpt(a) \leq end(s)$ and the definition of the restricted event stream. The same arguments hold for $\subset$.

Proof of Lemma 2. By induction on $q \in \mathsf{EQ}$. The two directions of the equivalence are proven simultaneously. In order to be able to apply the induction hypothesis, we need to generalize our statement from $\mathcal{E}$ to $\mathcal{E}\ |_b^e$ and will prove:

For all event queries $q \in \mathsf{EQ}$, all event streams $\mathcal{E}$, and all answers (s, Σ) we have

(a) $\forall b \leq begin(s) \forall e \geq end(s). \quad q \lhd_{\mathcal{E}|_b^e}\ s, \Sigma \quad \Longrightarrow \quad q \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}}\ s, \Sigma$, and

(b) $\forall b \leq begin(s) \forall e \geq end(s). \quad q \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}}\ s, \Sigma \quad \Longrightarrow \quad q \lhd_{\mathcal{E}|_b^e}\ s, \Sigma.$

We need to distinguish the same cases as in the definition of the semantics in Chapter 5. Albeit rather lengthy and mechanical here, induction the most natural way to prove any statement about something defined inductively and also gives us the chance to reflect our definition of semantics. Since we pretty much have only the inductive definition to work with, it is also questionable whether another way to prove the statement can be found at all.

We will only analyze the base case $q \in \mathcal{T}^q$ and the two representative and interesting inductive cases $q = \texttt{andthen}[[q_1, q_2]]$, $q = \texttt{without}\ \{q_1\}\ \texttt{during}\ \{q_2\}$ here. The analysis for the other inductive cases is either trivial, or follows a similar pattern and is pretty obvious, once one has seen the proofs for one or two representative cases.

In the following let $b \leq begin(s)$, $e \geq end(s)$ be arbitrary (they are all-quantified in the statement we wish to prove!).

Case $q \in \mathcal{T}^q$, i.e., q is an atomic event query.

(a) By assumption $q \lhd_{\mathcal{E}|_b^e}\ s, \Sigma$ and definition of semantics for the atomic case we have

(1) $begin(s) = end(s) = r$ and (2) $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\ |_b^e$.

We must show (i) $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\ |_r^r$.

This is simply the auxiliary proposition from above applied to (1) and (2).

(b) By assumption $q \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s, \Sigma$ and definition of semantics for the atomic case we have

(1) $begin(s) = end(s) = r$ and (2) $s = \langle d^r \rangle^r_r \sqsubset \mathcal{E}\,|^{end(s)}_{begin(s)} = \mathcal{E}\,|^r_r$.

We must show (i) $s = \langle d^r \rangle^r_r \sqsubset \mathcal{E}\,|^e_b$.

Again, this is just the auxiliary proposition applied to (1) and (2).

Case $q = \texttt{andthen}[[q_1, q_2]]$.

(a) By assumption and definition of semantics we have

(1) $q_i \lhd_{\mathcal{E}|^e_b} s_i, \Sigma_i$ for $i = 1, 2$, (2) $s' \sqsubset \mathcal{E}\,|^e_b$, and
(3) $begin(s) \leq begin(s_i)$, $end(s_i) \leq end(s)$ for i=1,2.

We must show (i) $q_i \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s_i, \Sigma_i$ for $i = 1, 2$ and (ii) $s' \sqsubset \mathcal{E}\,|^{end(s)}_{begin(s)}$.

Statement (ii) follows directly from the auxiliary proposition. For (i) apply the induction hypothesis (a) to (1) to obtain $q_i \lhd_{\mathcal{E}|^{end(s_i)}_{begin(s_i)}} s_i, \Sigma_i$ for $i = 1, 2$. Use of (3) and the induction hypothesis (b) on this yields (i). (Note that this is an instance where we needed the generalized statement of (b) instead of the simple right to left implication "$\Leftarrow$" in the original Lemma.)

(b) We have

(1) $q_i \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s_i, \Sigma_i$ for $i = 1, 2$, (2) $s' \sqsubset \mathcal{E}\,|^{end(s)}_{begin(s)}$, and
(3) $begin(s) \leq begin(s_i)$, $end(s_i) \leq end(s)$ for i=1,2.

We must show (i) $q_i \lhd_{\mathcal{E}|^e_b} s_i, \Sigma_i$ for $i = 1, 2$ and (ii) $s' \sqsubset \mathcal{E}\,|^e_b$.

Again, (ii) is just the auxiliary proposition on (2). For (i), an application of induction hypothesis (a) to (1) using fact (3) gives $q_i \lhd_{\mathcal{E}|^{end(s_i)}_{begin(s_i)}} s_i, \Sigma_i$. Now apply induction hypothesis (b) to get (i).

Case $q = \texttt{without}\ \{q_1\}\ \texttt{during}\ \{q_2\}$.

(a) We have

(1) $q_2 \lhd_{\mathcal{E}|^e_b} s, \Sigma$ and
(2) $\forall s' \subset \mathcal{E}\,|^e_b\ \forall \Sigma'.\ (begin(s) \leq begin(s') \wedge end(s') \leq end(s) \wedge q_1 \lhd_{\mathcal{E}|^e_b} s', \Sigma')$
$\Longrightarrow \Sigma\,|_V \cap \Sigma'\,|_V = \emptyset$.

We must show

(i) $q_2 \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s, \Sigma$ and
(ii) $\forall s'' \subset \mathcal{E}\,|^{end(s)}_{begin(s)}\ \forall \Sigma''.\ (begin(s) \leq begin(s'') \wedge end(s'') \leq end(s) \wedge q_1 \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s'', \Sigma'')$
$\Longrightarrow \Sigma\,|_V \cap \Sigma''\,|_V = \emptyset$.

For (i) we apply induction hypothesis (a) to (1). Now (ii): let $s'' \subset \mathcal{E}\,|^{end(s)}_{begin(s)}$ and Σ'' be arbitrary with $begin(s) \leq begin(s'')$, $end(s'') \leq end(s)$, and (*) $q_1 \lhd_{\mathcal{E}|^{end(s)}_{begin(s)}} s'', \Sigma''$. Since also

$s'' \sqsubset \mathcal{E}\,|_b^e$ (auxiliary proposition), it suffices to show $q_1 \lhd_{\mathcal{E}|_b^e} s'', \Sigma''$, then we can apply (2). But this is just induction hypothesis (b) for (*).

(b) We have

(1) $q_2 \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} s, \Sigma$ and

(2) $\forall s' \subset \mathcal{E}\,|_{begin(s)}^{end(s)}\ \forall \Sigma'.\ (begin(s) \leq begin(s') \wedge end(s') \leq end(s) \wedge q_1 \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} s', \Sigma')$
$\Longrightarrow \Sigma\,|_V \cap \Sigma'\,|_V = \emptyset.$

We must show

(i) $q_2 \lhd_{\mathcal{E}|_b^e} s, \Sigma$ and

(ii) $\forall s'' \subset \mathcal{E}\,|_b^e\ \forall \Sigma''.\ (begin(s) \leq begin(s'') \wedge end(s'') \leq end(s) \wedge q_1 \lhd_{\mathcal{E}|_b^e} s'', \Sigma'')$
$\Longrightarrow \Sigma\,|_V \cap \Sigma''\,|_V = \emptyset.$

We get (i) by applying induction hypothesis (b) to (1). Now (ii): let $s'' \subset \mathcal{E}\,|_b^e$ and Σ'' be arbitrary with $begin(s) \leq begin(s'')$, $end(s'') \leq end(s)$, and (*) $q_1 \lhd_{\mathcal{E}|_b^e} s'', \Sigma''$. Similar to above since also $s'' \subset \mathcal{E}\,|_{begin(s)}^{end(s)}$ (auxiliary proposition), it suffices to show $q_1 \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} s'', \Sigma''$, then we can use (2). Applying induction hypothesis (a) to (*) yields $q_1 \lhd_{\mathcal{E}|_{begin(s'')}^{end(s'')}} s'', \Sigma''$. Making use of $begin(s) \leq begin(s'')$, $end(s) \geq end(s'')$, and induction hypothesis (b) we are done.

6.5 Discussion

The main idea in the proof of our theorem has been to split the work into the two Lemmas.

Lemma 1 gives the bound β for all legal event queries by analyzing the required temporal restriction. Since our definition of legal event queries is not inductive, no induction is required in the proof.

Lemma 2 is a more general claim about event queries (whether legal or not). Its essence is that the beginning time $begin(s)$ and ending time $end(s)$ of an answer (s, Σ) to some event query q are such that only incoming atomic events happening between $begin(s)$ and $end(s)$ have been considered in the query evaluation of q. Events happening before or after have not and will not influence whether (s, Σ) is an answer or not.

Even if one did not care about legal event queries and bounded life-span, Lemma 2 still proves a useful property about our language. If a language construct were to violate Lemma 2, the language would become unsound: in its evaluation of a query to some answer we would consider events that happen outside the duration of the answer; in the worst case, we would have to consider events that might or might not happen in the future — and crystal gazing seems a little out of place for a good reactive language.

The separation of the proof of the theorem into two lemmas should also be beneficial when XChange's event query language is extended or revised. If we introduce a new language construct, say a new composition operator such as `meets` (suggested in [BBP04]), we only need to extend the proof of Lemma 2 with another case. If we change our notion of legal event queries, we only need to revise the proof of Lemma 1.

Chapter 7

Incremental Evaluation of Event Queries

Having described XChange's event queries both informally (Chapter 4) and formally (Chapter 5), we now turn to look at their evaluation.

We start by outlining requirements and considerations for the evaluation of event queries (Section 7.1). Evaluation of atomic event queries is conceptually pretty straightforward and we touch this only briefly (Section 7.2) before turning to the heart of this chapter, the evaluation of composite event queries (Section 7.3). Ideas on how to prove correctness w.r.t. the declarative semantics (Section 7.4) and on how to optimize event query evaluation (Section 7.5) round off this chapter.

7.1 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 need not check if the second had an answer previously and know that the composite `and`-query q is not answered, yet.

When 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; thus it avoids that cost of each single evaluation grows much. 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.[1]

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 tree 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.

[1] 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.

Utilization of Bounded Event Life-Span XChange's event query language has carefully been engineered, so that all legal event queries *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.

We have already seen that this is necessary to keep per event evaluation cost constant (cf. above). It is also necessary to keep storage requirements for events limited, as already discussed in Chapter 2.2.4.

Treatment of Variable Assignments As we have seen in the introduction of the event query language (Chapter 4) and the declarative semantics (Chapter 5), 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.

7.2 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 [BS02a]. We need not 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 7.5.

7.3 Evaluation of Composite Event Queries

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

7.3.1 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 [DG96, 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 an end 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 database in mind [GJS92a, GJS92b, GJS93].

  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 [GD93, GD94]. 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.[2] 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`

[2]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 is the way, this approach is described in [DG96, chapter 5.3]. We do however feel that this perspective is too limiting and does not do justice to tree-based composite event detection approaches in general. In the tree-based approach used in [CKAK94], for example, nodes do store events.

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 [CKAK94] and been used in a number of systems subsequently, e.g., GEM [MS97] and EPS [ME01].

The basic idea for this kind of incremental evaluation can be found in the *rete algorithm* [For82]. It describes an incremental forward-chaining evaluation for use in inference systems where over time new facts are told to the system.

7.3.2 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:

- **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 an implementation easy to debug.

  In contrast, an automaton constructed from a query bears little resemblance to the query structure. The construction is potentially error-prone and, in case of a bug in the implementation, we have to decide first whether the bug due to a faulty automaton construction or due to a bug in the execution of the automaton. 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 we will indicate in 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 [ME01]. 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 query length, leading to an expensive construction and bad memory requirements. (Lazy automaton construction can help here but usually makes implementation significantly harder.)

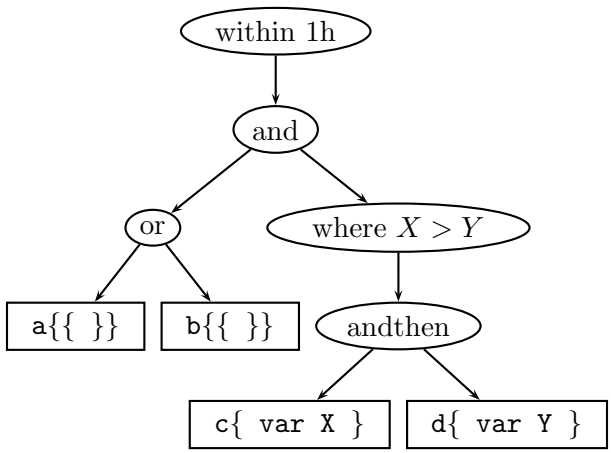
- **Versatility.** It has been shown that the tree-based approach can be adapted to various requirements. GEM [MS97] presents a version that can deal with delays in event reception due to unsynchronized clocks. EPS [ME01] presents a version that also deletes events when they time out. In this work we present a version that also supports deletion of events and, moreover, deals with variable assignments (cf. 7.1).

  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

Figure 7.1 A (composite) event query and its operator tree

```
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

Petri nets-based SAMOS [GD94] provides limited access to data contained in events, but does provide no easy way to accommodate substitutions or substitution sets arising from atomic event queries.)

7.3.3 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.[3] For example the query in Figure 7.1(a) has the operator tree in Figure 7.1(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` $\{q_2\}$. Further, 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 as discussed in Chapter 5.3.4.

In order to use the operator tree in an incremental evaluation, some inner nodes are given a storage for composite events (that is, tuples consisting of an event sequence and a substitution set). In this storage, the node stores all composite events it received from its children that might be needed in the future to compose an event. The storage depends 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`.

[3] 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 efficency. 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.

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 `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 paired up with all the stored events from the left child to form the composite events that are handed over to `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 paired up 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, the 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 storage of 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.

7.3.4 Bottom-Up Data-Flow for Event Detection

In the operator tree of 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 Chapter 5, 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 latter 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 former 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 $\Sigma\,|_{V_R} \subseteq \Sigma_R\,|_{V_R}$ 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.[4]

We already touched on this in Chapter 4.7.3: 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: "$\Sigma = \Sigma_L \wedge \Sigma_R$."[5] This view of

[4] 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.

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

substitution sets is beneficial if we need to accommodate negations (see Section 7.3.6): we can simply also have negated constraints $X \neq t$. Hence, the implementation of event query evaluation (see Chapter 8) 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 treatment 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 $end(s_L) < begin(s_R)$ 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 $end(s_L) = end(s_R)$ and hence automatically $end(s_L) \not< begin(s_R)$.

7.3.5 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.

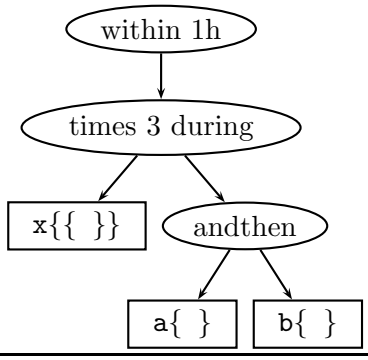
Event deletion depends on the current time and the life-span bound(s) provided by the event query. Time restriction operators (`in`, `before`,`within`) put a bound on all events stored in their subtrees in the operator tree. The same applies to the `during FiniteTimeInterval`-operators.

Accordingly, we traverse the operator tree top-down to delete events. We maintain an absolute time interval $[min...max]$ 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., $[min..max] := [-\infty..now]$ (where now is the current time and we have $end(s) \leq now$ for any event (s, Σ) stored in the tree). At each node in our traversal we do the following:

- 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 $[min'..max']$.

Figure 7.2 Operator tree for a `times...during...`-query



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 (see Chapter 6).

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 any 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

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

(operator tree depicted in Figure 7.2) 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.

7.3.6 Special Considerations: Duplicate Elimination, Partial Matches, Negation, Existential Quantification

The preceeding 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 Chapters 4 and 5.

Duplicate Elimination The evaluation algorithm discussed above can, in some rare cases, yield duplicate answers that should not be duplicate according to our formal semantics (Chapter 5). 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 are based on sets and relations and thus ask 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.

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 exists; we choose here to give it another child that matches any atomic event together with an event storage for that child. Keep in mind that, as pointed out earlier, only references to these atomic events need to be stored, so the memory consumption is acceptable.

Accordingly, the partial `andthen[[ ]]`-inner node has 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. Detection of events works the same way as in the total `andthen[ ]` case. However we do modify the answer that is passed to the parent node: those events from the middle child's storage that happen between the events from the left and the right child are added into 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, that in other words have both a non-defining (positive) occurrence and a defining (negative) occurrence, need special considerations. 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 (Chapter 5) and of course also the informal description (Chapter 4.7.5), 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. 7.3.4 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 Chapter 8) is based on constraints rather than substitution sets. It is especially convenient to use constraints since Xcerpt 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 our join operation $\bowtie$ from above (see Section 7.3.4) 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), ..., (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 $S := \{(s_1, \Sigma_1), (s_2, \Sigma_2), \ldots, (s_N, \Sigma_N)\}$ which satisfy the conditions given by the semantics of the `times`-operator (cf. 5.3.9). 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 ask "$\subseteq$", not "$=$" in condition (2).) If D is an absolute time interval, simply set $\Sigma := \Sigma'$.

Extension of this idea to `times atmost` and `times atleast` is trivial. 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.

7.4 Correctness of the Incremental Event Query Evaluation

Since the algorithm works incrementally, proving correctness is not so easy. But we can simplify the problem can by dividing it into two:

- First we forget that the algorithm works incrementally and stores events to avoid recomputation. We pretend that all incoming events (with occurrence times from time point 0 of the query registration to the current time point t) are processed in one single evaluation. That is, we use the operator tree to detect all composite events happening between 0 and t.[6] In contrast, the incremental algorithm detects only composite events occurring at time t,[7] all prior events have already been detected in previous evaluations.
- Second we prove that for an evaluation at time point t, the operator tree of the incremental algorithm will have stored all events that can possibly play a role as constituent events in a composite event with occurrence time t. Proving this requires two things:
  - we need to show that the bottom-up query evaluation stores all events that might be needed later on in the right storages of the right operator nodes, and
  - we need to show that event deletion does not delete any events that might be needed later on. (Most work for this has already been done with Chapter 6).

7.5 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

[6] That is, $0 \leq \mathit{begin}(s)$ and $\mathit{end}(s) \leq t$ for a composite event (s, Σ).

[7] This is, $\mathit{end}(s) = t$ for a composite event (s, Σ).

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.[8] 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 [BFO04]. 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.

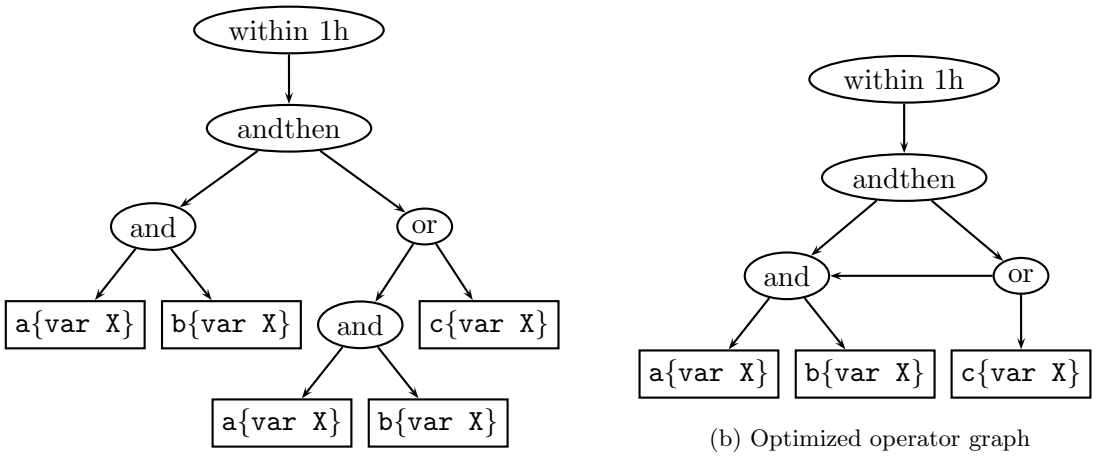
A stream-based evaluation of Xcerpt queries (i.e., atomic event queries) would provide a great optimization for the event query evaluation in XChange. Surely concepts from stream-based evaluation of other query languages, esp. XPath [BBC+03], can be borrowed for this. For a stream-based evaluation of 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 [BCD+05, Olt05], 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

[8] Keep in mind that during the composite event query evaluation (processing of inner nodes in the operator tree) t need not be accessed anymore; it suffices to maintain a reference to it when constructing the event sequences that are part of the answers. Thus t can be moved to secondary memory after the evaluation of all atomic event queries).

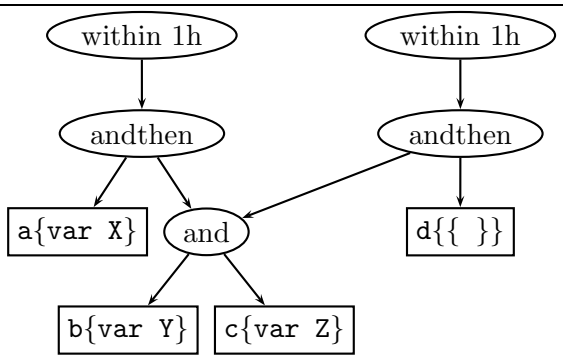
Figure 7.3 Optimization of an operator tree into an operator graph



(a) Unoptimized operator tree

(b) Optimized operator graph

Figure 7.4 Optimization of composite event queries with shared sub-expressions



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 [ME01].

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

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

depicted in Figure 7.3(a) can be optimized to the graph in Figure 7.3(b).

For another example, the composite event queries

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

and

```
andthen {
    and { b{var Y}, c{var Z} },
    d{{ }}
} 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 7.4.

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 later registered query 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.[9]

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. For example in relational algebra a linear join $((A \bowtie B) \bowtie C) \bowtie D$ can by means of associativity be rewritten to a bushy join $(A \bowtie B) \bowtie (C \bowtie D)$.

In principle, restructuring of the operator tree as an optimization techniques 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 less laws that can be used for restructuring in XChange's event query language that in say relational algebra. 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 allow as 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.

[9] 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.

Still, there is one simple optimization that we can learn from relation 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 is used on a more complicated expression, it is often possible to push it into 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 [
    and {
        a{ var X },
        b{ var Y }
    },
    c {var Z}
] where { var X < var Y }
within 1h
```

to

```
andthen [
    and {
        a{ var X },
        b{ var Y }
    } where { var X < var Y },
    c {var Z}
]
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 and cannot move the selection around himself. 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 implementing 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 used 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 on a normal 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.

Chapter 8

Implementation

The XChange prototype, which is currently being developed, comprises three major modules that correspond to the three parts of an Event-Condition-Action (ECA) rule in the language XChange:

- the Event module, which receives incoming event messages and evaluates event queries on them;
- the Condition module, which evaluates Web queries; and
- the Action module, which executes updates and raises new event messages.

In the scope of this thesis, the Event module for the XChange prototype has been developed. Its heart is a proof-of-concept implementation of the event query evaluation described in the previous chapter. The source code is available on the Web at `http://purl.oclc.org/NET/xchange`.

When considering control flow in the whole XChange prototype, we can see that the Event module takes a prominent and controlling position: it receives incoming events and initiates any activity in the Condition and Action module. Therefore, with the development of the Event module, also the basic architecture for the whole XChange prototype has been developed.

The Event module has been implemented in Haskell [Pey02, Tho99], a choice that largely depended on the availability of a Haskell implementation of Xcerpt. Since the Condition and Action module will also depend on Xcerpt, Haskell will also be the language of choice to implement these. The prototype uses some extensions to the Haskell language provided by the Glasgow Haskell Compiler [GHC], to allow parallelism and interprocess communication between the modules.

We will obviously not discuss every aspect of the implementation, but try to give a high-level view that allows one to quickly grasp the source code structure and find the right places to make changes or extensions. We start by giving an overview of XChange's architecture (Section 8.1). We then explain reception and handling of incoming events (Section 8.2), implementation of the event query evaluation (Section 8.3), and implementation of the event deletion (Section 8.4). Next, instructions for building XChange from the source code and

running XChange are given (Section 8.5). Finally, we indicate unimplemented aspects of the XChange's Event module and ideas for future extensions (Section 8.6).

8.1 An Architecture for XChange

The three major modules of XChange take care of evaluating event queries, evaluating condition queries, and executing actions, respectively. They run as separate threads; each thread's main loop is implemented as a tail-recursive function (the standard way to program an infinite loop in Haskell):

- `eventHandlerLoop` in `XChange.Event.EventHandler`
- `conditionHandlerLoop` in `XChange.Condition.ConditionHandler`
- `actionHandlerLoop` in `XChange.Action.ActionHandler`

All three functions have a type `XChangeSetup -> State -> IO ()`; note however that the definition of `State` is different in each module, it comes from `XChange.Event.State`, `XChange.Condition.State`, `XChange.Action.State`, respectively.

The three threads (or rather functions) communicate using *Channels*, a Concurrent Haskell extension [PGF96]. Channels provide a buffered First In, First Out (FIFO) messages-passing communication between threads. Due to the read and write operations on the channels, the result type of the three functions above is the `IO ()`-monad. The names of the channels are provided by the first parameter `XChangeSetup` (from `XChange.Data.XChangeSetup`); this parameter also supplies some other setup information such as functions to print debugging output.

The channel `eventChannel` supplies incoming event messages to `eventHandlerLoop`. The channel `conditionChannel` is used to signal successful event query evaluations from `eventHandlerLoop` to `conditionHandlerLoop`. Similarly, the channel `actionChannel` is used to signal successful condition query evaluations from `conditionHandlerLoop` to `actionHandlerLoop`.

The parameter `State` is used to maintain the system's state, that is, what XChange rules are registered and, in the Event module's case, the partial event query evaluations. Leaving the first parameter `XChangeSetup` aside, the three functions follow the basic pattern

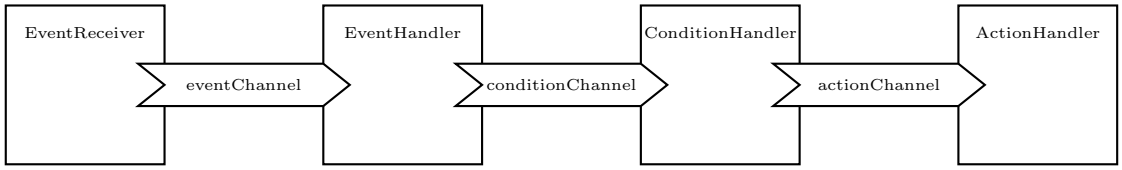
$$\text{loop}(\textit{state}) = \text{loop}(\text{computeNewState}(\textit{state})).$$

We are only interested in modifying the state and performing side-effects (captured by the result type being `IO ()`) such as communication along the channels.

Additionally to these three threads, a fourth thread takes care of the reception of events, that is, opening network sockets, parsing the incoming text, etc. It is part of the Event module and implemented in the function `eventReceiverLoop` in `XChange.Event.EventReceiver`.

This basic architecture with four threads communicating by means of three channels is depicted in Figure 8.1.

Figure 8.1 The basic XChange architecture: four threads (depicted as boxes) communicating via channels (depicted as arrows)



8.2 Event Reception and Handling

The Event Receiver thread (`XChange.Event.EventReceiver`) receives incoming event messages from simple TCP/IP connections (default port 4711), one message per connection. It then parses them (using Xcerpt's parsing facilities), augments them with the receiver-supplied management information `xchange:reception-time` and `xchange:reception-id`, and finally writes them to the channel `eventChannel` for the Event Handler thread.

For testing and debugging purposes, there is also an alternative Event Receiver `XChange.Event.AlternateEventReceiver` which reads incoming event messages from files instead of the network. The filenames can be supplied on the command line when starting the XChange program; each file has to contain exactly one event message in XML. Since shorter event message are easier to read when debugging, this alternative Event Receiver does not require the standard message envelope with root `xchange:event` and does not augment management information like `xchange:reception-time` and `xchange:reception-id`.

The Event Handler thread (`eventHandlerLoop` in `XChange.Event.EventHandler`) reads parsed event messages (`AtomicEvent` in `XChange.Data.Event`) from `eventChannel`. Its second parameter of type `State` (defined in `XChange.Event.State`) supplies a list of all present partial event query evaluations (`PartialEventQueryEval` in `XChange.Event.PEQE`).

For each of the partial event query evaluations, the Event Handler now calls first the event query evaluation (cf. Section 8.3). Event query evaluation returns a new partial event query evaluation[1] and a list of composite events (i.e., answers to the composite event query found in the current evaluation) of type `CompositeEvent` (defined in `XChange.Data.Event`). By writing these composite events to the channel `conditionChannel`, they are handed over to the Condition Handler thread (`XChange.Condition.ConditionHandler`).[2]

Next, the Event Handler calls the event deletion function (cf. Section 8.4) on the partial event query evaluation returned from the event query evaluation. Event deletion returns the final partial event query evaluation that becomes part of the `State` for the next evaluation step on the next incoming event message. It is not strictly necessary to perform event deletion after each event query evaluation; we have chosen this for convenience of implementation and ease

[1] Keep in mind that Haskell is a pure functional language, so we cannot simply "update" the present partial event query evaluation but need to return a new one as result of a function call.

[2] Note that the Condition Handler supplied in the framework of this thesis is only a dummy component that does not actually evaluate conditions. It simply serves to demonstrate the architecture (cf. Figure 8.1). Its implementation is a separate, ongoing project outside the scope of this thesis.

of testing and debugging. A more advanced implementation could perform event deletion asynchronous from query evaluation, e.g., in periodical time intervals or whenever certain memory limits are reached.

8.3 Event Query Evaluation

The data type `PartialEventQueryEvaluation` (in `XChange.Event.PEQE`) represents the storage-augmented operator tree described in Chapter 7. Let's look at the first lines of its definition:

```
type PEQEqueue = (PartialEventQueryEval, [CompositeEvent])
[...]
data PartialEventQueryEval
  = PAtomic Term
  | PAnd PEQEqueue PEQEqueue
[...]
```

The type constructor `PAtomic` represents an atomic event query, the only leaf type in the tree; `Term` is the corresponding type for an Xcerpt query term. The type constructor `PAnd` represents an `and`-node in the tree. It has partial event query evaluations as left child and right child and additionally for each child an event storage of type `[CompositeEvent]` (a list of composite events).

The function `evaluateQuery` in `XChange.Event.EventQueryEvaluation` performs the actual event query evaluations. It traverses the operator tree by recursively calling itself. As its first parameter it takes the incoming atomic event and as its second parameter the partial event query evaluation to be evaluated. It returns the updated partial event query evaluation together with a list of composite events. The returned composite events signify successful query evaluations that are to be handed over to the parent node in the operator tree (or, in case of the root node, to be written to the channel `conditionChannel` by the Event Handler).

The function `evaluateQuery` uses pattern matching on the partial event query evaluation to decide which operator node is being evaluated. The basic pattern for inner nodes is (compare also Chapter 7.3.4):

1. Recursively call `evaluateQuery` on the children; this returns a list of composite events and the updated partial event query evaluations of the children.

2. Try to compose composite events from the component composite events returned by the recursive call in 1. and those stored in the current node.

3. "Update" the event storage, i.e., build a new partial event query evaluation from the partial event query evaluations returned in 1. by children, the current storage, and the composite events returned in 1. by the recursive calls to the child nodes.

4. Return the composite events composed in 2. (empty list if none were found) and the "updated" partial event query evaluation from 3.

Composing composite events (step 2. above) from component composite events (successful evaluations, "firings," of the current node's children) is implemented in function names `joinFiringsAnd`, `joinFiringsAndthen`, etc. Instead of working with substitutions sets, we exploit —as pointed out in Chapter 7— the analogy of substitutions and (disjunctive normal form) constraints. In the case of an `and`-node, for example, we do not compute a join $\Sigma = \Sigma_1 \bowtie \Sigma_2$, but take the constraint $C = C_1 \wedge C_2$ and solve it into disjunctive normal form. This is convenient since we can simply use Xcerpt's constraint solver (wrapper functions are provided in `XChange.Data.XChangeXcerpt`).

Our implementation of `evaluateQuery` can return duplicate composite events (cf. Chapter 7.3.6). The Event Handler compensates for this by calling `duplicateRemoval` (in `XChange.Event.EventHandler`) on the result returned by `evaluateQuery`.

8.4 Event Deletion

Deletion of events with expired life-span, as described in Chapter 7.3.5, is implemented by `deleteExpiredEvents'` in `XChange.Event.EventDeletion`. It traverses the partial event query evaluation in a top-down manner and constructs a new partial event query evaluation in which the expired events have been deleted. The function takes four parameters: the current time (`now`), the lower bound of the current restriction interval (`tmin`), the upper bound of the current restriction interval (`tmax`), and the current partial event query evaluation. It returns the new, "cleaned" partial event query evaluation.

Time points are represented by the data type `XChangeTime`, durations (lengths of time) are represented by `XChangeDuration`; together with auxiliary functions these reside in `XChange.Data.XChangeTime`.

8.5 Instructions for Building and Running XChange

Source Code Layout The top directory of the source distribution contains the following files and subdirectories:

- file `compile.sh`: a shell script to compile XChange;
- file `Main.hs`: it implements XChange's main routine, i.e., parses command line parameters and starts up the four threads depicted in Figure 8.1;
- directory `terms/`: it contains test cases for various language constructs, i.e., XML documents to be received by the Event Receiver;
- directory `Xcerpt/`: it contains the required Xcerpt source code; the current version of XChange has been built and tested with Xcerpt version 0.9-1;
- directory `XChange/`: it contains the source code for XChange, i.e., the modules described in the previous sections.

Building XChange XChange has been tested and built with the Glasgow Haskell Compiler [GHC] version 5.04.3. GHC provides its own make facility, so there is no need to use a `Makefile` or something similar to compile XChange; you can simply call `ghc` directly on `Main.hs`. The shell script `compile.sh` contains the necessary command line parameters for `ghc`. In a Unix shell simply do

```
> ./compile.sh Main
> mv Main xchange
```

to build XChange.

A parser for XChange rules is currently being developed separately from this thesis and, at the time of writing this, not yet available and integrated. Therefore, event queries can currently only be compiled statically into the XChange code. The file `XChange/Event/MyQueries.hs` contains exemplary event query definitions and can be extended for new event queries. To "register" these queries, one must modify the list of initially registered event queries `initialState` in `XChange/Event/State.hs`. For example the definition

```
initialState = State [(generatePEQE myPartialQuery,0)]
```

will register the event query

```
andthen [[
    and { a{var X}, b{var X}}
    and { c{var Y}, d{var Y}}
]]
```

as event query of the XChange rule with identifier 0.

Command Line Parameters XChange provides extensive debugging output. Command line parameters allow to enable it and redirect it into files:

- `-r` displays all received event messages;
- `-i` displays all partial event query evaluation after the event query evaluation, but before the event deletion (the "intermediate" state);
- `-c` displays all partial event query evaluation after the event deletion (the "cleaned" state not containing any expired events);
- `-f` displays all composite events obtained from successful query evaluations, i.e., all composite events that are to be forwarded to the Condition Handler (the "firings");
- `-d` enables auxiliary debugging output.

Each of these parameters can be supplied with a filename, e.g., `-r`*filename*, to redirect to output from the terminal in to a file. Output has XML format to allow automatic processing, e.g., to visualize partial event query evaluations as trees.

A list of files can be supplied as parameters; if this is done, XChange does not receive events from the network but simply reads them from the files (cf. Section 8.2). This is convenient for testing and debugging.

Example: the following command starts XChange on the test terms supplied in `terms/terms_partialandthen/` and enables displaying of the partial event query evaluations after the event deletion as wall as displaying of all composite events obtained from successful event query evaluations:[3]

```
> ./xchange -g -f terms/terms_partialandthen/*.txt
```

8.6 Unimplemented Aspects and Future Work

We now discuss unimplemented aspects of XChange's event query language and directions for future work on the current implementation.

HTTP Event Reception For the future it is envisioned that XChange receives (and also sends) event messages via HTTP or other Web protocols. Unfortunately, library support for HTTP in Haskell is rather limited and every existent library forces its own control flow model upon a program. Hence, the prototype uses simple TCP/IP socket communication instead. A simple approach to realize HTTP support without having to modify the Haskell prototype would be to implement (e.g., in Java where the necessary extensive and stable libraries are available) an adapter that receives event messages by HTTP and forwards them on a TCP/IP connection to XChange.

Timer Events The prototype implements `without`, `times`, and `of` only in the forms

```
without    { EQ } during { CEQ }
times Mult { EQ } during { CEQ }
Mult of    { EQ } during { CEQ }
```

where the duration interval (right hand side) is given by a composite event query. It does not implement the forms

```
without    { EQ } during [b .. e]
times Mult { EQ } during [b .. e]
Mult of    { EQ } during [b .. e]
```

where the duration interval (right hand side) is given by a specification of an absolute time interval. In principle, the latter can be reduced to the former, but this means that we have to generate event messages at time points b and e. Even if we don't try it with this reduction but implement directly, we still need to generate some event (in the sense of some event of the programming system, not an XChange event) at time point e, which causes the evaluation

[3]The test terms are named `01.txt`, `02.txt`, and so on. The wildcard `*` should be used only with a shell sorting the filenames resulting from the wildcard expansion into alphabetical order, e.g., the Bash shell.

of the `without-`, `times-`, or `of`-node in the operator tree (b is less important since nothing happens at that time point). Haskell library support for generating an event at a given time point is very limited; the only possibility offered in GHC is access to Posix signals (via the library `System.Posix.Signals`). This has portability issues and is quite difficult to use without introducing race conditions (i.e., the order of events with respect to their occurrence times can get mixed up). We have preferred portable and clean code for the prototype over a full implementation of all language features.

Rule Registration The prototype has built in facilities to register new XChange rules it is being sent. Rule registration depends, of course, on parsing of rules. As mentioned above, the parser for XChange rules is still under development and not yet integrated. Hence, the function `parseRule` in `XChange.Data.Rule` is only a dummy that is to be replaced by a proper implementation when the parser is finished. Also note that the implemented rule registration does not consider issues of authorization and authentication (cf. also Chapter 2.4.1) and "blindly" registers any incoming rule.

Further Operators With the exception of the operators requiring timer events (see above), the prototype implements all composition operators and temporal restrictions described in Chapters 4 and 5. However, with further development of the language XChange, definition of new operators is not unlikely. We therefore provide a checklist indicating where the current source code needs to be extended to implement a new operator:

- The data type `EventQuery` in `XChange.Data.EventQuery` represents syntax trees of event queries and needs an additional case for a new operator.
- Similarly, the storage-augmented operator tree data type `PartialEventQueryEval` in `XChange.Event.PEQE` needs an additional case.
- The function `generatePEQE` in `XChange.Event.PEQE` implements the translation from the syntax tree `EventQuery` to the storage-augmented operator tree `PartialEvent-QueryEval`.
- The function `peqeshow` in `XChange.Event.PEQE` is used to display `PartialEventQuery-Eval`-objects for debugging purposes.
- The function `evaluateQuery` in `XChange.Event.EventQueryEvaluation` needs an additional case to implement evaluation of the new operator (cf. Section 8.3).
- The function `deleteExpiredEvents'` in `XChange.Event.EventDeletion` needs an additional case to implement deletion of events for the new operator (cf. Section 8.4).

Obviously, the last two extensions are the substantial ones requiring the most work.

Chapter 9

Conclusions

9.1 Summary

This work has dealt with querying events for reactivity on the Web. We have presented the event query language that is part of the reactive rule-based language XChange (Chapter 4). In XChange, (atomic) events are represented as XML documents and communicated in a push-manner. They are queried with event queries, which serve the double purpose of specifying when to react to some event and extracting data from events. XChange supports composite event queries, that is, event queries that detect temporal patterns of atomic events.

Semantics for event queries have been described using a novel approach that is justified by the duality of answers as event sequences and substitution sets (Chapter 5). We have used the semantics to offer a formal proof that all legal event queries need only events with a bounded life-span to be evaluated correctly (Chapter 6).

We have developed an incremental algorithm to evaluate event queries (Chapter 7). A prototype of the event query evaluation has been implemented as a component for the prototype of the complete XChange language (Chapter 8).

9.2 Related Work

We have already discussed related work directly in the various chapters covering the aspects to the language, its semantics, and its evaluation. Thus, we only give a brief summary here and indicate the sections where a more detailed explanation is to be found.

A number of event query languages with support for composite events have been developed in the past, mainly in the active databases community, for example: COMPOSE, [GJS92a, GJS92b, GJS93], SAMOS [GD93, GD94], Snoop [CKAK94], REACH [BZBW95], GEM [MS97], EPS [ME01], and Amit [AE04]. Features of these languages differ significantly, and even the interpretation of similar looking operators (e.g., the sequence operator) can be unexpectedly different. Attempts to compare and unify various languages have been made in [ZU99]. We have pointed out similarities and differences between XChange's event query language and other languages when introducing the language constructs informally in Chapter 4. The most important difference is that XChange considers data contained in events by

means of finding substitutions for free variables contained in event queries.

Our formal, declarative semantics for composite event queries in XChange provided in Chapter 5 follow a novel approach. The prime feature of our approach is the accommodation of data in event messages —captures by substitutions for free variables— even in a context of advanced composition operators such as negation or quantifications. Previous work on semantics for composite event queries such as [GJS93, CKAK94] ignores the issue of extracting data from event messages. In SAMOS [GD94] data in events is considered (though not to the extend of XChange); however only operational semantics obtained from a transformation of event queries into special Petri nets are provided. The semantics provided for XChange event queries in Chapter 5 are inspired by and based on the declarative semantics of Xcerpt [Sch04]; they diverge however in that instead of giving rules to find a minimal model, we work with a predetermined event stream and build event sequences that are part of the answer (additionally to substitution sets).

Three methods for the (incremental) evaluation (also called detection) of composite event queries have been suggested in prior work: finite state automata, Petri nets, and operator trees (or graphs) with a bottom-up data flow of events. Of these, the tree-based approach seems to be the most widely used. For reasons discussed in Chapter 7.3, we use a tree-based approach to evaluate event queries in XChange. Our method extends upon previous tree-based approaches in that it has to consider substitution sets as part of the bottom-up data flow.

9.3 Future Research Directions

Our work on event queries and composite event detection in XChange gives rise to new directions for future research. These roughly fall into the tree categories of language design, semantics, and event query evaluation.

9.3.1 Language Design

Further Language Constructs In this work, we have formally defined and implemented a coherent set of composition operators and temporal restrictions for XChange event queries. Though our set of language constructs has a wide range of applicability and allows easy specifications of many types of composite events, we cannot say that it is complete; ideas for further language constructs are outlined in [BBP04]. In this work we only realized a limited set of constructs that have very intuitive semantics (also when variables are present) and go together well. In future it is desirable to add further constructs, especially if use cases (see below) call for it.

Underpinning with Use Cases To make the event query language presented in this work and the whole XChange language a mature reactive language, it needs underpinning with use cases. Use cases should motivate the existing language constructs, possibly give rise to refinements, and serve to identify constructs that might be missing. In doing this, use cases can also serve as a practical measure for the completeness of the language. An account of use cases for reactivity and evolution on the Web in general can be found in [ABB+05a] and

[ABB+05b]. Work on more specific use cases for XChange has, at the time of writing this, already started.

Advanced Calendar and Time Specifications The current XChange uses a very simple notion of time: time points specified as date and time, absolute time intervals specified by two time points, and relative time intervals (durations) specified by a number of seconds, minutes, hours, days etc. For the future it is envisioned to integrate XChange with a system that allows more advanced time specifications such as specifications in different calendar systems (e.g., Gregorian calendar, Hebrew calendar), culture or location specific times (e.g., Western versus Orthodox Easter), or periodic intervals (e.g., working-day). A primary candidate for this is the Calendar and Time Type Systems (CaTTS) [BRS05].

Visual Language for (Composite) Event Queries In Chapter 3.1.5 we have shortly mentioned visXcerpt, a visual language for specifying Xcerpt rules. Its rendering of Xcerpt query terms is directly applicable as a visual rendering of XChange's atomic event queries. To render composite event queries, we are still lacking suitable renderings for composition operators and temporal restrictions. Visual renderings for these should reflect the intuition behind the language constructs.

Of course, we can take this further and not only render atomic and composite event queries but render whole XChange event-condition-action rules.

XChange for the Semantic Web Currently, XChange concentrates on data of the standard Web, i.e., raw XML data. This is true for both data of event messages and data of Web resources. However, mechanisms to query and reason with Semantic Web data such as RDF data or OWL ontologies in the query language Xcerpt are currently being investigated. XChange embeds Xcerpt and thus "inherits" such mechanisms almost automatically.

Though the event query language of XChange is based on Xcerpt by using its notion of query terms, an issue that has not yet been investigated is a possibility to provide a form of rule chaining or "event views." At the moment, event queries directly query the incoming event data; it is not possible to apply transformations to the incoming events and then query the transformed data, i.e., to provide logical views over the event data and query these. Investigating such chaining or view mechanisms for events is highly relevant if event data is RDF and comes as an XML serialization of RDF: extensions to query RDF with Xcerpt rely on rule chaining [FBB05] and thus are applicable to XChange event queries only if a similar facility is provided there.

9.3.2 Semantics

Reduced and Less Complicated Semantics The current declarative semantics are maybe a little complicated and cumbersome since they involve four items: queries, the event stream, event sequences, and substitution sets. The intricacy of the semantics is necessary, however, to accommodate advanced event query constructs such as free variables, partial specifications, or negation. Still, it might be worth investigating reduced sets of our event query language for which semantics can be defined in an easier fashion and are thus easier to

understand for users. This is especially interesting if such a reduced set finds application in a domain such a mobile devices where support for the full language could exceed memory or processor speed limits.

Theoretical Measures for Language Completeness and Expressiveness There are currently almost no theoretical measures for the completeness or expressiveness of a composite event query language. The only popular measure is an analogy to regular expressions: the stream of incoming events is understood as a word (or string) in which we look for substrings specified by a regular expression and an event query respectively.

This is not really a good measure, however, especially for a reactive language on the Web: the most basic operator for regular expressions is the composition, which indicates that two characters (which correspond to atomic events) or two substrings (which correspond to composite events) follow each other directly. This would correspond to events coming in direct succession, i.e., with no other events between them. On the Web this makes barely sense if we assume that anyone can send an event message and thus by purpose or accidentally interleave events that are specified to come in direct succession.

Investigating other and better measures for the completeness and expressiveness of an event query language, such as different forms of temporal logics, seems a worthwhile task not only for XChange but research on composite events in general.

Extensions and Modularity The current semantics do not accommodate event consumption or event instance selection (cf. 3.3.4). They also rely on a very simple notion of time supporting only time points, absolute (non-periodic) time intervals specified by two time points, and relative time intervals of fixed length.

Accommodating features such as event consumption or instance selection, or allowing advanced time specifications, e.g., with CaTTS (see above), require some extensions to the current semantics. Preferably, such extensions should come in some form of "modular add-on" to the standard semantics. Doing this probably also requires some (non-substantial) modifications of the current semantics in order to allow such modular extensions.

Integration with XChange Semantics This work has provided semantics for the event query language of XChange, which is part of the XChange language as a whole. To reason about XChange programs, we need not only semantics for event queries but for complete event-condition-action rules. Hence, the event query semantics will become a part of the whole XChange semantics and need to be "integrated." In principle, this should pose no big problem. For matters of elegancy and automation of proofs about program properties, investigating restricted subsets of the event query language and modular semantics can still be interesting.

9.3.3 Event Query Evaluation

Interaction with Transaction Management In this thesis and also in work on XChange in general, transaction management is treated as somewhat of a side issue. For a deployment of XChange in real world situations, however, transaction management is very important.

For the influence of transaction management on (composite) event detection consider the following situation: As part of some transaction, a rule at site A fires and raises an event e. This event e is sent to site B. When the transaction now is canceled we have to issue a roll-back. If the event e is treated as part of the transaction we need to aborted all effects e has had or might have somewhere on the Web. This means taking back all updates of Web data that were initiated by e, either as sole atomic event or as part of a composite event, deleting all semi-composed composite events containing e, and maybe also firing rules that have not fired because of e, e.g., rules having an event query that contains a `without` e.

We can see that transaction management has considerable influence on event query evaluation, but also whole XChange rules. Issues concerning from transaction management should be investigated in future for XChange. Note however that our assumption to ignore transaction management first to limit the scope of this work and XChange in general is not invalidated by a need for transaction management. It should be possible to add the necessary mechanisms without having to perform major modifications in the existing system.

Formal Correctness Proof In Chapter 7.4, we have given an idea on how to prove correctness of our event query evaluation algorithm w.r.t. the declarative semantics of Chapter 5. Though outside the scope of this thesis, working out the proof completely and formally is of course desirable.

Optimizations, Efficiency, Complexity We have outlined some ideas on how to optimize both atomic event query evaluation and incremental composite event detection in Chapter 7.5. These would be interesting to investigate in future together with experimental results on their efficency and maybe also theoretical analysis of their complexity.

9.4 Conclusion

Reactivity on the Web is an emerging research field, and processing of events on the Web is one of its core issues. This work has taken a novel view on events, event queries and composite event detection: Events are represented by XML data, and event queries serve not only to specify composite events that require a reaction, but also to extract data in the form of variables assignments from events. A rich set of language constructs to specify composite events has been defined. The language constructs have been provided with declarative semantics and evaluation algorithms that both take into consideration substitutions for the free variables contained in an event query.

Appendix A

Grammar for XChange Event Queries

The following provides a context-free grammar for XChange's event query language in a simple EBNF notation.

A.1 EBNF notation

The EBNF notation used here follows the conventions of the XML 1.1 Recommendation [BPS+04, Section 6]:

- Rules are denoted in the form `symbol ::= expression`.
- Symbols (names of non-terminals) are written as plain text, the first letter is always capitalized.[1]
- Literal strings (sequences of terminals) are written in double quotes `"literal"`, or in single quotes if a double quote is part of the literal `'with "double"quotes'`.
- Classes or ranges of characters are written in square brackets; `[a-zA-Z]` matches any lowercase or uppercase letter of the alphabet.
- Parentheses are used as grouping construct `(expression)`.
- Concatenation is simply written with a whitespace `A B`, and takes higher precedence than alternation.
- Alternation is expressed with a vertical bar `A | B`.
- Optionality is expressed with a question mark; `A?` matches `A` or nothing.
- Repetition is expressed with a plus sign or an asterisk; `A+` matches one or more occurrences of `A`, `A*` matches zero or more occurrences of `A`.

[1] This is a slight deviation from the notation of [BPS+04] used in order to enhance readability.

- Comments are written C-style `/* comment */`.
- Ellipses are written with three dots ....

A.2 Event Queries

```
EQ  ::= CEQ | AEQ

AEQ ::= /* Xcerpt Query Term (with optional where clause)*/
      | AEQ "in" FiniteTimeInterval
      | AEQ "before" TimePoint
      | "var" VarName "->" AEQ

CEQ ::= CEQ "in" FiniteTimeInterval
      | CEQ "before" TimePoint
      | CEQ "within" Duration
      | "and"     "{"  EQ ("," EQ)*  "}"
      | "or"      "{"  EQ ("," EQ)*  "}"
      | "andthen" "["  EQ ("," EQ)+  "]"
      | "andthen" "[[" EQ ("," EQ)+  "]]"
      | "without" "{" EQ "}" "during" "{" CEQ "}"
      | "without" "{" EQ "}" "during" FiniteTimeInterval
      | "times" Mult ("any" VarList)? "{" EQ "}" "during" "{" CEQ "}"
      | "times" Mult ("any" VarList)? "{" EQ "}" "during" FiniteTimeInterval
      | Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                                  "during" "{" CEQ "}"
      | Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                                  "during" FiniteTimeInterval
      | "var" VarName "->" CEQ
      | CEQ "where" "{" Condition ("," Condition)* "}"

Condition ::= /* see Chapter 4.5.4 of [Sch04] */

VarList ::= var VarList (, var VarName)*

VarName ::= [a-zA-Z] [a-zA-Z0-9]*

Mult    ::= (atleast|atmost)? NatNum

NatNum  ::= [1-9] [0-9]*
```

A.3 Legal Event Queries

```
LEQ ::= AEQ
      | CEQ "in" FiniteTimeInterval
      | CEQ "before" TimePoint
      | CEQ "within" Duration
      | "without" "{" EQ "}" "during" FiniteTimeInterval
      | "times" Mult ("any" VarList)? "{" EQ "}" "during" FiniteTimeInterval
      | Mult "of" ("any" VarList)? "{" EQ ("," EQ)* "}"
                                                "during" FiniteTimeInterval
```

A.4 Time Specifications

```
FiniteTimeInterval ::= "[" TimePoint  ".." TimePoint  "]"

TimePoint ::= Year "-" Month "-" Day "T"
              Hour ":" Min (":" Sec ("." Frac)?)? (TZD)?
Year      ::= "0000" | ... | "9999"      /* four digit year */
Month     ::= "01" | ... | "12"          /* two digit month */
Day       ::= "01" | ... | "31"          /* two digit day */
Hour      ::= "00" | ... | "23"          /* two digit hour */
Min       ::= "00" | ... | "59"          /* two digit minute */
Sec       ::= "00" | ... | "59"          /* two digit second */
Frac      ::= [0-9]+
TZD       ::= Z | ("-"|"+") Hour ":" Min

Duration  ::= (NonNegNum "y")? (NonNegNum "d")?
              (NonNegNum "h")? (NonNegNum "m")?
              (NonNegFloat "s")?
                 /* constraint: string non-empty!*/

NonNegNum   ::= [0-9]+
NonNegFloat ::= [0-9]+   ( "." [0-9]+ )?   ( "e" ("+"|"-")? [0-9]+ )?
```

Appendix B

Document Type Definition for Event Messages

The following is an exemplary Document Type Definition (DTD) [BPS+04] for the event messages used in XChange. When using DTDs and namespaces together it is necessary to commit oneself to a fixed namespace prefix and give the fully qualified name for each element [BHLT04]; the prefix chosen for the XChange namespace (`http://pms.ifi.lmu.de/xchange`) here is `xchange:`.

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

<!ATTLIST xchange:event
    xmlns:xchange CDATA #FIXED "http://pms.ifi.lmu.de/xchange" >

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

In the element definition of `xchange:event`, we use a parameter entity in place of the free content inside an event message. This parameter entity has to be defined application dependent. For example the messages about flight cancellations used in Chapter 3.3 (Figure 3.7 on page 28) could continue our DTD with the following definition of the parameter entity:

```
<!ENTITY % content "(flight-cancellation)">
```

```
<!ELEMENT flight-cancellation (
    flight-number,
    passenger ) >

<!ELEMENT flight-number (#PCDATA)>
<!ELEMENT passenger     (first, (middle)?, last)>
<!ELEMENT first         (#PCDATA)>
<!ELEMENT middle        (#PCDATA)>
<!ELEMENT last          (#PCDATA)>
```

Appendix C

Notation used for Semantics

C.1 Atomic Events

Notation	Explanation
$a = d^r$	An atomic event a. The event message is a data term $d \in \mathcal{T}^d$ and it is received at time point $r \in \mathbb{T}$
$rcp(a) = r$	The reception time r of an atomic event $a = d^r$.
$\mathcal{T}^d$	The data term domain, i.e., the set of all conceivable data terms (or XML documents). Formally defined in [Sch04].
$(\mathbb{T}, \mathbb{D})$	The time domain, which contains time points $\mathbb{T}$ and durations $\mathbb{D}$ (e.g., $(\mathbb{T}, \mathbb{D}) \cong (\mathbb{N}, =, <, 0, +, -)$).
$[b..e]$	A time interval lasting from time point $b \in \mathtt{T}$ until time point $e \in \mathtt{T}$. Formally, $[b..e] = \{t \in \mathbb{T} \mid b \leq t \leq e\}$.

C.2 Substitution Sets

Notation	Explanation
Σ	A substitution set, i.e., a set of substitutions σ.
$\sigma = \{X \mapsto \sigma(X), \ldots\}$	A substitution, i.e., an assignment of values $\sigma(X) \in \mathcal{T}^d$ to variable names X. More formally: a partial function $\sigma : V \to \mathcal{T}^d$. Note that this is a simplified definition that suffices in the context of event queries; see [Sch04, chapter 7] for a more sophisticated definition.
$\Sigma \mid_U$	Restriction of Σ to the variable set U. Formally, $\Sigma \mid_U = \{\sigma' \mid \exists \sigma \in \Sigma\ \forall x.\ \sigma'(x) = \sigma(x)$ if $x \in U,\ \sigma'(x) = \perp$ otherwise $\}$.
V	The set of all variables having at least one *negative* (defining) occurrence in a given event query q. Negative occurrences are simply said those occurrences that are not inside a `without`-construct.

C.3 Event Sequences

Notation	Explanation
$s = \langle a_1, \ldots a_n \rangle_b^e$	An event sequence, i.e., a sequence of temporally ordered atomic events a_i together with a beginning time $b \in \mathbb{T}$ and an ending time $e \in \mathbb{T}$. Note that $rcpt(a_i) \in [b..e]$ for all i.
$begin(s) = b$	The beginning time of an event sequence $s = \langle a_1, \ldots a_n \rangle_b^e$.
$end(s) = b$	The ending time of an event sequence $s = \langle a_1, \ldots a_n \rangle_b^e$.

C.4 Stream of Incoming Events

Notation	Explanation
$\mathcal{E} = \langle a_1, a_2, \ldots a_n \rangle_b^e$	The stream of incoming events contains *all* atomic events a_i received by some XChange-aware Web site (or some query at an XChange-aware Web site) in the time interval from b to e. It is a sequence where the atomic events are *ordered* by their reception time.

C.5 Operations on Event Sequences

Notation	Explanation
$s \subset s'$	Event sequence s is a *subsequence* of event sequence s', i.e., all atomic events in the event sequence s are also contained in s'.
$s \sqsubset s'$	Event sequence s is a *complete subsequence* of event sequence s', i.e., the atomic events in the event sequence s are also contained in s' and their sequence in s' is not "interleaved" by other atomic events. For example: $\langle w, x, y \rangle_b^e \sqsubset \langle v, w, x, y, z \rangle_{b'}^{e'}$, but $\langle w, y \rangle_b^e \not\sqsubset \langle u, v, x, y, z \rangle_{b'}^{e'}$ (due to the "interleaving" x).
$s'' = s \cup s'$	The *union* of event sequences s and s', i.e., s'' contains all atomic events that are in s or in s'. Note that $begin(s'') = \min\{begin(s), begin(s')\}$ and $end(s'') = \max\{end(s), end(s')\}$.
$\bigcup_{1<i<n} s_i$	Shorthand for $s_1 \cup \ldots \cup s_n$.

C.6 Relating Queries and Answers

Notation	Explanation
q	An event query, e.g., $q =$ `and` $\{ q_1, q_2 \}$ `within 1h`.
(s, Σ)	A (potential) answer to a composite event query is a tuple consisting of a sequence of atomic events s and a substitution set Σ.
s, Σ	Alternative notation for (s, Σ). To ease readability parentheses are usually dropped.
$q \lhd_{\mathcal{E}} s, \Sigma$	Event query q is *answered* by (s, Σ) under the event stream $\mathcal{E}$. Note that the parentheses around s, Σ have been dropped for readability. The rule corresponding to the event query q is triggered once for each answer (s, Σ) with a *maximal* substitution set Σ.

Appendix D

On the Choice of Event Sequences as Answers to Event Queries

In XChange, the answer to a composite event query consists of

- a sequence of all atomic events that must have occurred in order to answer the query, and
- a substitution set giving assignments for the free variables contained in the query.

The need for a substitution set is obvious. It is also obvious that we should have means to see which atomic events lead to a successful answering of the event query. It is however not obvious that these should be captured in a flat sequence.

Using event sequences is not the only possibility to define a notion of answers for composite event queries. In XChange, we first also investigated the possibility to define answers as instances of composite queries with the atomic event queries replaces by the matching event instances [BBP04]. As a composite query has a tree structure (inner nodes are composition operators, leafs are atomic event queries), this can be given a straightforward representation as XML document, and this representation reflects the temporal pattern specified in the query. For example, the event query

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

(using again the shorthand notation introduced in Section 4.3, page 48, `a[[ ]]`, `b[[ ]]`, `c[[ ]]` are atomic event queries) could have the answer

```
event-andthen {
    event-and {
        a[ "1" ],
        b[ "2" ]
    },
    c [ "1" ]
}
```

where `a[ "1"]`, `b[ "2"]`, `c[ "1"]` are atomic events answering `a[[ ]]`, `b[[ ]]`, `c[[ ]]`, respectively.

While it is quite nice that the answer to a composite event query mirrors the structure of the query, we found that using flat event sequences as a notion for answers is better:

- The flat structure is quite simple, a nested structure is more complicated and should only be chosen if it offers significant benefit.
- The definition of an answer as an event sequence requires no knowledge of the structure and operators used in queries. In contrast, a the nested, query structure mirroring answer needs a definition for every single operator, e.g., to bring together the operator `and` and it representation in an answer as `event-and`. While the representation is straightforward for the simple operators like `and`, it is not for more advanced operators like `times`, and especially not for `without` (since it detects non-occurrence, there is no event message to put inside `without`!).
- Event sequences, due to their similarities with sets, allow a relatively easy definition of declarative semantics.
- Finally, is seems to us that event sequences lead to more intuitive semantics.

To illustrate the last point, consider the (admittedly somewhat artificial) query

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

as an example. Using event sequences as answers, it has when evaluated on the stream

```
< a[ "1" ], b[ "2" ] >
```

of incoming events, only the one answer:

```
event-seq [
    a[ "1" ],
    b[ "2" ]
]
```

This is what one probably would expect intuitively. Using the discussed tree structure that mirrors the query structure could however lead to two answers:

```
event-and {
    event-or {
        a[ "1" ]
    },
    event-and {
        a[ "1" ],
        b[ "2" ]
    }
}
```

and also

```
event-and {
    event-or {
        b[ "2" ]
    },
    event-and {
        a[ "1" ],
        b[ "2" ]
    }
}
```

Of course we could redefine that answer in a way that we do not obtain these two answers but only one by collecting all possibilities for answering the `or` (i.e., we would have only

```
...
    event-or {
        a[ "1" ],
        b[ "2" ]
    },
...
```

instead of the two `event-or`s). This however would surely just lead to other counterintuitive examples and cause difficulties with finding the right substitution sets.

Taking up on the analogy of event sequences as answers and minimal models for formulas (see Section 4.6), asking for an answer that mirrors the event query structure is a bit similar to asking for a proof of a logic formula: there can be several proofs for a single formula.

Bibliography

[ABB+05a] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Nicola Henze, Wolfgang May, Paula-Lavinia Pătrânjan, and Michael Schroeder. Use-cases on evolution: Deliverable I5-D2. Technical report, 6th Framework Programme project REWERSE number 506779 (see `http://rewerse.net`), February 2005.

[ABB+05b] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Nicola Henze, Wolfgang May, Paula-Lavinia Pătrânjan, and Michael Schroeder. Use-cases on reactivity: Deliverable I5-D3. Technical report, 6th Framework Programme project REWERSE number 506779 (see `http://rewerse.net`), February 2005.

[ABC+99] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. XSL Transformations (XSLT) Version 1.0. W3C recommendation, World Wide Web Consortium (W3C), November 1999.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, first edition, 1999.

[AE04] Asaf Adi and Opher Etzion. Amit — the situation manager. *The International Journal on Very Large Data Bases (VLDB Journal)*, 13(2):177–203, 2004.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[BBC+03] 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/`.

[BBEP05] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Event queries in XChange. Technical report, Institute for Informatics, University of Munich, May 2005. Submitted to *Special Issue of the Computer Networks journal on Web Dynamics.*

[BBP04] James Bailey, François Bry, and Paula-Lavinia Pătrânjan. Composite event queries for reactivity on the Web. Technical Report PMS-FB-2004-26, Institute for Informatics, University of Munich, 2004.

[BBSW03] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 1053–1056. Morgan Kaufmann, September 2003.

[BCD+05] 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.

[BCF+03] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C working draft, World Wide Web Consortium (W3C), November 2003. `http://www.w3.org/TR/2003/WD-xquery-20031112/`.

[Ber03] Sacha Berger. Conception of a graphical interface for querying XML. M.Sc. thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2003.

[BFB+05] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web reconsidered: Design principles for versatile Web query languages. *International Journal of Semantic Web and Information Systems (IJSWIS)*, 1(2), 2005.

[BFM98] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, August 1998. `ftp://ftp.rfc-editor.org/in-notes/rfc2396.txt`.

[BFO04] François Bry, Tim Furche, and Dan Olteanu. Datenströme. *Informatik Spektrum*, 27(2):168–171, 2004.

[BFPS04] François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data retrieval and evolution on the (Semantic) Web: A deductive approach. In *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning 2004 (PPSWR 2004)*, volume 3208 of *Lecture Notes in Computer Science*. REWERSE, Springer-Verlag, September 2004.

[BFS00] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *The International Journal on Very Large Data Bases (VLDB Journal)*, 9(1):76–110, 2000.

[BG04] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C recommendation, World Wide Web Consortium (W3C), February 2004.

[BHLT04] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.1. W3C recommendation, World Wide Web Consortium (W3C), February

2004.
`http://www.w3.org/TR/2004/REC-xml-names11-20040204`.

[BM01] James Bailey and Szabolcs Mikulás. Expressiveness issues and decision problems for active database event queries. In *Proceedings of the 8th International Conference on Database Theory (ICDT 2001)*, pages 68–82. Springer-Verlag, January 2001.

[BP05] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005)*. ACM Press, March 2005.

[BPS+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1. W3C recommendation, World Wide Web Consortium (W3C), February 2004.
`http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[BPW02] 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.

[BRS05] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar types and constraints for Web applications. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*. ACM Press, May 2005.

[BS02a] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation Unification. In *Proceedings of the International Conference on Logic Programming (ICLP 2003)*, volume 2401 of *LNCS*, August 2002.

[BS02b] François Bry and Sebastian Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Web, Web-Services, and Database Systems: NODe 2002 Web and Database-Related Workshops (Proceedings of the 2nd Annual International Workshop of the Working Group "Web and Databases")*, volume 2593 of *LNCS*. German Informatics Society (GI), Springer-Verlag, October 2002.

[BS03] François Bry and Sebastian Schaffert. An entailment relation for reasoning on the Web. In *Proceedings of Rules and Rule Markup Languages for the Semantic Web 2003 (RuleML 2003)*, volume 2876 of *LNCS*. Springer-Verlag, October 2003.

[BSS04] François Bry, Sebastian Schaffert, and Andreas Schröder. A contribution to the semantics of Xcerpt, a Web query and transformation language. In *Proceedings of the 18th Workshop on (Constraint) Logic Programming (WLP 2004)*, March 2004.

[BZBW95] Alejandro P. Buchmann, Jürgen Zimmermann, José A. Blakeley, and David L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proceedings of the 11th International Conference on Data Engineering (ICDE 1995)*, pages 117–128. IEEE Computer Society Press, March 1995.

[CD99] James Clark and Steve DeRose. XML path language (XPath) version 1.0. W3C recommendation, World Wide Web Consortium (W3C), November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2001.

[CKAK94] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 606–617. Morgan Kaufmann, September 1994.

[CT04] John Cowan and Richard Tobin. XML information set (second edition). W3C recommendation, World Wide Web Consortium (W3C), February 2004.

[DG96] Klaus R. Dittrich and Stella Gatziu. *Aktive Datenbanksysteme: Konzepte und Mechanismen*. International Thomson Publishing, first edition, 1996.

[DS04] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs); draft-duerst-iri-11. Technical report, Internet Engineering Task Force (IETF), November 2004.
`http://www.w3.org/International/iri-edit/draft-duerst-iri-11.txt`.

[FBB05] Tim Furche, François Bry, and Oliver Bolzer. XML perspectives on RDF querying: Towards integrated access to data and metadata on the Web. In *Tagungsband zum 17. GI-Workshop über Grundlagen von Datenbanken (Proceedings of the 17th GI-Workshop on the Foundations of Databases)*, pages 43–47. Institute of Computer Science, Martin-Luther-University Halle-Wittenberg, May 2005.

[For82] Charles L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[GD93] 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.

[GD94] 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.

[GHC] The glasgow haskell compiler.
`http://www.haskell.org/ghc/`.

[GJS92a] 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.

[GJS92b] Narain H. Gehani, H.V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD*

International Conference on Management of Data (SIGMOD 1992), pages 81–90, San Diego, California, June 1992. ACM Press.

[GJS93] 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.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[HV02] Annika Hinze and Agnès Voisard. A parameterized algebra for event notification services. In *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME 2002)*, pages 61–63. IEEE Computer Society Press, 2002.

[Int00] International Organization for Standardization. *ISO 8601:2000. Data elements and interchange formats — Information interchange — Representation of dates and times*. International Organization for Standardization, Geneva, Switzerland, 2000.

[ME01] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1):1–10, 2001.

[MM04a] Frank Manola and Eric Miller. RDF primer. W3C recommendation, World Wide Web Consortium (W3C), February 2004.
`http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[MM04b] Frank Manola and Eric Miller. RDF primer. W3C recommendation, World Wide Web Consortium (W3C), February 2004.
`http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[MS97] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalised event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web ontology language overview. W3C recommendation, World Wide Web Consortium (W3C), February 2004.
`http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[Olt05] Dan Olteanu. *Evaluation of XPath Queries against XML Streams*. Ph.D. thesis (Doktorarbeit), Institute of Informatics, University of Munich, 2005.

[Pey02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, December 2002.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL 1996)*, pages 295–308. ACM Press, 1996.

[RC] The ruleCore® system — advanced business situation detection. `http://www.rulecore.com`.

[SB04] Sebastian Schaffert and François Bry. Querying the web reconsidered: A practical introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004*, August 2004.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Ph.D. thesis (Doktorarbeit), Institute for Informatics, University of Munich, 2004.

[Sta95] Ryan Stansifer. *Theorie und Entwicklung von Programmiersprachen*. Prentice-Hall, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the 6th International Conference on Database Theory (ICDT 1997)*, volume 1186 of *LNCS*, pages 19–40. Springer-Verlag, January 1997.

[WW97] Misha Wolf and Charles Wicksteed. Date and time formats. W3C note, World Wide Web Consortium (W3C), September 1997. `http://www.w3.org/TR/1998/NOTE-datetime-19980827/`.

[ZU99] Detlef Zimmer and Rainer Unland. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE 1999)*, pages 392–399. IEEE Computer Society Press, March 1999.