The Language XChange: A Declarative Approach to Reactivity on the Web

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

von

Paula-Lavinia Pătrânjan

28. Juli 2005

Erstgutachter:	Prof. Dr. François Bry (Universität München)
Zweitgutachter:	Prof. Dr. Sara Comai (Politecnico di Milano)
Tag der mündlichen Prüfung:	23. September 2005

Abstract

The research topic investigated by this thesis is *reactivity on the Web*. Reactivity on the Web is an emerging research issue covering: updating data on the Web, exchanging information about events (such as executed updates) between Web sites, and reacting to combinations of such events. Following a declarative approach to reactivity on the Web, a novel *reactive language* called *XChange* is proposed. Novelties of the language are represented by the proposed data metaphor intended to ease the language understanding and the supported reactive features tailored to the characteristics of the Web. Realising this pressuposed refining, extending, and adapting to a new medium some of the concepts on which active database systems are built upon.

Reactivity is specified in XChange by means of reactive rules (or event-condition-action rules) having the following components: the event part is a query against events that occurred on the Web, the condition part is a query against Web resources (expressed in the Web query language Xcerpt), and the action part is a transaction specification (specifying updates to be executed and events to be raised in an all-or-nothing manner). Novel in XChange is its ability to detect *composite events* on the Web, i.e. possibly time related combinations of events that have occurred at (same or different) Web sites.

XChange introduces a novel view over the Web data by stressing a clear separation between *persistent data* (data of Web resources, such as XML or HTML documents) and *volatile data* (event data communicated on the Web between XChange programs). Based on the differences between these kinds of data, the data metaphor is that of *written text* vs. *speech*. XChange's language design enforces this clear separation and entails new characteristics of event processing on the Web.

After motivating the need for a solution to reactivity on the Web, this thesis introduces the design principles and syntax of the language XChange accompanied by use cases for demonstrating the practical applicability of its constructs. Important contributions of the thesis are the specification of the language semantics and the description of an algortihm for evaluating XChange programs.

Zusammenfassung

Diese Dissertation untersucht das Forschungsthema *Reaktivität im Web*, ein neu entstehendes Forschungsgebiet, das Änderungen von Daten im Web, Austausch von Ereignisdaten zwischen Websites, sowie Reaktion auf (atomare und zusammengesetzte) Ereignissen einbezieht. Einem deklarativen Ansatz folgend, schlägt diese Dissertation die neuartige *reaktive Sprache XChange* vor. Neuerungen der Sprache sind eine neue Datenmetapher, die das Verständnis von XChange erleichtern soll, und die Features der Sprache, die auf das Web zugeschnitten sind. Konzepte der aktiven Datenbanken wurden verfeinert, erweitert und angepasst um dem neuen Medium (das Web) gerecht zu werden.

Reaktivität ist in XChange durch reaktive Regeln, so genannte Ereignis-Bedingung-Aktion Regeln, spezifiziert: Der Ereignisteil ist eine Anfrage bzgl. Ereignissen die im Web auftreten. Eine Neuerung in XChange ist die Fähigkeit zusammengesetzte Ereignisse (eventuell zeitbezogene Kombinationen von Ereignissen, die an den gleichen oder verschiedenen Websites aufgetreten sind) im Web zu erkennen. Der Bedingungsteil ist eine Anfrage an Web-Dokumente (die in der Web-Anfragesprache Xcerpt spezifiziert ist). Der Aktionsteil spezifiziert Transaktionen (Änderungen die durchgefürt werden sollen und Ereignisse die gesendet werden sollen).

Durch die hervorgehobene Trennung zwischen *persistenten Daten* (Web-Dokumente wie zum Beispiel XML oder HTML Dokumente) und *flüchtigen Daten* (Ereignisdaten die im Web zwischen XChange Programmen ausgetauscht werden), führt XChange eine neuartige Sicht auf die Daten im Web ein. Eine Metapher für die strenge Unterscheidung zwischen persistenten und flüchtigen Daten findet sich in der natürlichen Sprache wieder: die Unterscheidung zwischen *geschriebenem* und *gesprochenem Wort*. Das Sprachdesign erzwingt diese klare Trennung und verursacht neue Charakteristika der Ereignisverarbeitung im Web.

Diese Dissertation führt nach der Begründung der Erfordernisse für Reaktivität im Web die Designprinzipien und die Syntax der Sprache XChange ein. Es werden Anwendungsfälle, die die Benutzung und Relevanz der Sprachkonstrukte zeigen, vorgestellt. Wichtige Beiträge dieser Dissertation sind die Spezifikation der Semantik von XChange und die Beschreibung der zur Auswertung von XChange Programmen nötigen Algorithmen.

Acknowledgements

I have been fortunate enough to have had the support of many people during the time spent at the Institute for Informatics, University of Munich. Many of them have contributed directly to the reactive language XChange presented in this thesis, and everyone has helped me to get where I am today.

It is a pleasure to express my gratitude to my supervisor, *Prof. Dr. François Bry* (Universität München), who has guided my path towards this thesis; owing to his advice and warnings I haven't followed dead-end paths and managed to complete this thesis in less than three years. I also thank him for the countless hours we have spent discussing research issues related to the work presented in this thesis.

I would like to thank *Prof. Dr. Sara Comai* (Politecnico di Milano), the second reviewer of this thesis, for her time and dedication to reading my thesis, her comments and suggestions for improvements. I am also grateful for her collaboration in organising the Workshop 'Reactivity on the Web', to be held at the International Conference on Extending Database Technology (EDBT 2006) in Munich.

I am obliged to *Prof. Dr. Helmut Schwichtenberg* (Universität München) and the other professors of the PhD programme "Logic in Computer Science" (Graduiertenkolleg Logik in der Informatik) I am part of for giving me the possibility to investigate interesting research issues and to work in the PhD programme's team.

During the last (almost) three years I have enjoyed many interesting discussions and the excellent working environment at the Research and Teaching Unit 'Programming and Modelling Languages', Institute for Informatics, University of Munich. I would like to thank our technical and administrative staff for providing me with help during this thesis. I would also like to stress the contributions made by some of the fellow researchers and students.

Dr. Norbert Eisinger, senior researcher in our group, has supported me during the time spent in Munich. I thank him for many helpful discussions.

Dr. Sebastian Schaffert, now a researcher at Salzburg Research, Austria, has developed the (Semantic) Web query language Xcerpt that is integrated into XChange. Without his work and collaboration, the language XChange wouldn't be that mature today. I also thank him for his help in typesetting this thesis.

Michael Eckert, now a fellow researcher, has written his master's thesis under the supervision of Prof. Dr. François Bry and myself. I thank him for fruitful discussions that helped improving this thesis and his collaboration on refining and implementing the XChange event query language. I also thank him for the careful reading of the most important chapters of this thesis and for its constructive criticism.

The time spent on writing this thesis would not have been so valuable without knowing *Sacha Berger*, who has worked with me on the visual counterpart of XChange. I thank him for his help and practical advice.

I have been fortunate to have *Tim Furche* as fellow researcher. With his broad spectrum and many interesting research ideas, he has provided me with many helpful discussions. I thank him for his collaboration on writing a scientific article on XChange.

Inna Romanenko, a graduate student working on her master's thesis under the supervision of Prof. Dr. François Bry and myself, investigates the use of XChange rules for business process modelling. I thank her for the good work on the EU-Rent case study.

Oliver Friedmann, student assistant in our group, has worked with me on the XChange prototype implementation. I thank him for integrating Xcerpt and XChange implementations and for his patience to restructure and comment the whole source code.

I would also like to acknowledge the support of *Sebastian Kraus* and *Bernhard Lorenz*. Both of them have shared an office with me, and I thank them for tolerating my stressed out phases.

Being member of the Network of Excellence "Reasoning on the Web with Rules and Semantics" (REWERSE) I have had the chance to work with researchers from different universities. I thank *Prof. Dr. José Júlio Alferes* (Universidade Nova de Lisboa), *Prof. Dr. James Bailey* (University of Melbourne), and *Prof. Dr. Wolfgang May* (Universität Göttingen) for their collaboration on project deliverables, scientific articles, and on organising the Workshop 'Reactivity on the Web'.

During my travels for attending conferences and workshops I have met a number of great people who have been cordially and helpful. I would like to especially thank *Andreas Doms* (Technische Universität Dresden), *Michael Kinateder* (Universität Stuttgart), *Prof. Dr. Christoph Koch* (Universität des Saarlandes), *Steffen Mazanek* (Universität der Bundeswehr München), *Thorsten Staake* (Universität St.Gallen), *Ioan Toma* (Digital Enterprise Research Institute Innsbruck), and *Dr. Laurenţiu Vasiliu* (Digital Enterprise Research Institute Galway). They have given me advice and encouraged me to go for what I want.

My parents, *Giorgeta and Mircea Pătrânjan*, deserve special thanks as they have supported and encouraged me during the last 27 years. They have always shown me the right way in life, even if this has implied to leave Romania for doing my PhD. I also thank my brother, *Cosmin*, and all my friends for being always there when I needed someone to talk to.

Last but not least, I would like to thank *Hans J. Kroner* for sustaining me patiently during the last two and a half years and for tolerating my workaholic lifestyle during the last six months. He has given me the energy I needed for completing this thesis; I couldn't have made it without him.

I have been very lucky to be founded for the last three years by the German Foundation for Research (Deutsche Forschungsgemeinschaft, DFG, `http://www.dfg.de`). The received grant has given me the possibility to investigate interesting research issues within the PhD programme "Logic in Computer Science" (Graduiertenkolleg Logik in der Informatik, `http://www.math.lmu.de/~gkli/`).

Also, as member of the 6th Framework Programme project "Reasoning on the Web with Rules and Semantics" (REWERSE, `http://rewerse.net`), project number 506779, I have received additional financial support from the European Commission and the Swiss Federal Office for Education and Science.

Paula-Lavinia Pătrânjan
Munich, 28th July 2005

CONTENTS

Part I

Introduction

CHAPTER
ONE

Motivation and Outline

1.1 Context

The work presented in this thesis has as context the World Wide Web (or simply, the Web) as it is today,

- a distributed system that builds upon the Internet, a network of different kinds of computer systems, nodes (like personal computers with or without server abilities or powerful computers acting as servers);
- a set of technologies implementing the model of interconnected pieces of information;
- a huge amount of data found on these computer systems, part of it available for reading or modifying by other nodes, and data that is communicated between nodes (a node's data may depend on the data found at other nodes).

Web technologies have been developed with different goals and tailored to different application areas, from security of data transmission to e-commerce. Thus, Web applications help already humans in resolving everyday life problems like managing appointments or booking flights and hotels. However, current Web technologies do have limitations that preclude an easier, less time consuming, less human interaction-based realisation of tasks. There are several research directions that are being or need to be investigated in order to bridge the existing recognised gaps. Having the same goal of easing people's life, the work of the thesis focuses on a single research direction. Before describing this research work, some of the notions used throughout this thesis are introduced here, as their comprehension provides a basis for understanding the contributions of this thesis and their practicability. *Note* that some of the notions have a slightly different meaning in the literature.

Notion of Web Resource. Web resource denotes data at a Web node that can be addressed through a URI (Uniform Resource Identifier [145]) or in the future through a IRI (Internationalised Resource Identifier [89]). This thesis assumes that the data found at a particular Web resource is in an XML-like format [159]. This is not to be understood as a restriction of the thesis' approach, even if some Web resources' data are represented using the relational model.

- Many Web-based applications use XML as data storage and data exchange format, as

  - XML has been recognised as a suitable format for data with no regular structure. Most data on the Web do not necessarily have a schema to which to conform to. However, data having a fixed structure can be also easily represented in XML.

  - the XML format has been established as *the* data interchange format.

- Future Web applications will surely be based on a combination of XML and ontology description languages. The later are intended to be used by computer systems to *understand* the *meaning* of data. The future Web – the *Semantic Web* – has already began to be “built” by developing ontology description languages and systems working with them, but there is still much work to be done in this respect.

- Ontology description languages do have XML serialisations and thus can be processed in the same manner as XML data can. For example, for RDF [148] data many XML serialisations have been proposed (such as RDF/XML [163]).

Notion of Web Site. Usually, a Web site is a set of interlinked Web pages (e.g. written in HTML [147]) that are available on the Web. Web sites offering services (such as an airline’s Web site with online booking abilities) have underlying programs that implement these services and database(s) containing the needed information (like flight timetables and reservation data). Database can be for example a Tamino [1] database or one or more XML documents. The notion of *Web site* is used here to denote actually the reactive program(s) (i.e. capable to detect and automatically react to happenings) that underly a Web site. Web sites without reactive abilities are not of interest in this context and thus the discussion does not refer to them.

Notion of Evolution. Many resources on the Web are dynamic, i.e. they can change their content over time. The need for changing (updating) data on the Web has several reasons: New information comes in calling for insertion operation of new data. Information is out-of-date calling for deletion and replace operations on data. Such changes on Web resources’ data need to be mirrored in changes to Web resources whose data depends on the initial changes (in other words, updates need to be propagated over related Web resources). *Evolution of data* on the Web comprises updating Web resources’ data and propagating updates on the Web. Thus, one can differentiate between

- *local evolution* that refers to updating data (inserting new data, deleting data, or replacing data) of a Web resource, and

- *global evolution* that refers to updating data of Web resources as consequence to remote updates (updates performed at other Web sites). *Note* that global evolution subsumes local evolution.

Notion of Reactivity. *Reactivity* on the Web is the ability of Web sites to detect happenings of interest that have occurred on the Web and to automatically react to them. Happenings of interest can be delays or cancellations of flights, or new discounts for flights offered by an airline. Example of reactions to such happenings are notifying colleagues about delays, looking for and booking another flight, or booking flights from a particular airline. In this thesis it is considered that reactivity subsumes evolution. Local evolution can be realised to some extent without reactive capabilities. But, global evolution needs reactivity in order to propagate updates over related Web resources. Moreover, reactions like booking of flights call for local and/or global evolution (see Section 1.2.1).

This thesis proposes *a novel solution to reactivity on the Web*. The need for finding such a solution has been already recognised (e.g. in [12, 102, 11]) as a *must-have* ability of the actual Web. The next section motivates the practicability of reactive capabilities through application scenarios. These scenarios reveal a set of basic concepts (Section 1.3) that are needed for understanding the proposed solution for reactivity on the Web.

1.2 Motivating Application Scenarios

This section describes the *Travel Organisation* use case, the intention being that of determining through simple life examples the current stage of Web technologies and the exigency of new technologies or better solutions to some of them.

1.2.1 Travel Organisation

Planning and booking travels, staying up to date with changes in the plan (e.g. due cancellations of flights or overbooking of hotels), and (if necessary) adapting to such changes is a time consuming task. E-business has made planning and booking of travels, e.g. a researcher's trip to a conference, a lot easier. One can now search for flight or train connections, for suitable hotels, can also compare offers on the Web and *book* with his or her credit card.

Informally, it is clear what under *booking* (a hotel, a flight, etc.) everyone understands. But, let's look behind the curtain to understand what booking on the Web involves: booking means changes of (i.e. updates to) one or more Web resources. For example, booking a flight on the Web represents inserting into the airline's documents the information regarding the desired flight (number, date, departure city, arrival city) and the person who is going to travel (name, dietary requirements, credit card information). Moreover, the number of free seats for the respective flight needs to be changed. In the case that the booking of flight has been done through a travel agency, changes need to be realised on some of the travel agency's documents, but they also need to be propagated to the airline offering the flight.

The task of organising travels implies two subtasks to be accomplished: initial planning and adapting plans to changes, which are detailed in the following sections. Existing Web-based applications support the subtask of initial planning but this still requires a lot of human interaction. However, reasoning-capable applications do not have the capability to detect situations requiring an adaptation of the initial plan.

Initial Planning

The *initial planning* subtask of organising travels consists of two phases:

1. gathering the information about transportation, overnight stays etc. required for the travel, and developing itineraries (a travel plan) based on this information, and

2. arranging (booking) the travel according to this plan.

1. Gathering Information and Plan Trip Generally, when planning a travel one knows the location(s) he/she wants to visit and a time period for doing this. In most cases the time period is not fixed at time points level, i.e. one might return late in the evening or next day in the morning, but it is likely to be constrained by working hours or appointments. Thus, the gathering of information for planning a travel involves finding flights and train connections, finding suitable hotels, considering weather forecast, and/or planning entertainment.

Another issue that plays a role in planning a travel is the amount of money that one is willing to pay for all the arrangements. This presupposes searching for cheapest accommodation and transportation means that conforms one's expectation. Moreover, limited time discounts can also be taken into consideration. *Note* that such discounts might not be always stored as Web resources; instead, they can be sent as notifications by Web-based information services.

2. Arranging the Travel According to Plan Gathering all the needed information on e.g. flights, train connections, hotels, and prices, represents the premises for planning a travel. After settling for a plan by choosing transportation means, accommodation, and (perhaps) entertainment, one needs to arrange the travel according to the plan, i.e. book flights, reserve seats in trains, make hotel reservations, buy tickets.

Booking a flight on the Web, for example, does not necessarily entail a successful execution of the desired reservation. Possible reasons for failures include system down-times, network communication problems, or problems caused by concurrency and time-delays, e.g. if the last seat on a flight was sold while one is still planning. As a consequence, reservations that are 'related' should be executed in an *all-or-nothing* manner. E.g. a hotel reservation without a flight reservation is useless. *Note* that the order in which such reservations are realised needs also to be taken into consideration; usually one might want to realise the flight reservation before the hotel reservation.

Scenario Mrs. Smith uses a travel organiser that plans her trips and reacts to happenings that could influence her schedule. One of the travel organiser's tasks is to plan Mrs. Smith's vacation in Provence, France. Mrs. Smith wants to visit Orange, Arles, Nîmes, and Marseilles. The trip should begin on 5th of March 2005 and end on 20th of March 2005. Carrying out the task of initial planning for this trip presupposes:

1. Gathering of and reasoning with information

- Searching for suitable flights from Munich to Lyon and back. Schedules and price tables of several airlines have to be queried and compared. Only flights departing on 5th of March 2005 and arriving on 20th of March 2005 before 21:30 o'clock are of interest. Moreover, the search is constrained by a price limit of EUR 400.
- Querying schedules and prices for train or flight connections for Lyon – Orange – Arles – Nîmes – Marseilles – Lyon. Like searching for flights between Munich and Lyon, queries need to have time and price constraints.
- Querying hotel reservation services' data to find suitable hotels in the cities Mrs. Smith wants to visit. Suitable refers to the quality of services (e.g. at least 2 stars), the prices (e.g. price per night should be cheaper than EUR 70), the time period for which single rooms are available for booking (e.g. Mrs. Smith wants to book a hotel in Lyon from 5th of March 2005 until 8th of March 2005), and the location of the hotels (e.g. a quiet area near to a metro station).
- Receiving notifications from different kind of information systems, like weather forecast services, or services announcing exhibitions. Reasoning with notifications' data and data of Web resources is needed for planning departures and arrivals but also for planning entertainment. For example,
  - if a notification is received informing that between 11st and 14th of March 2005 in Arles is going to rain, the vacation could be planned so as to leave for Arles on 15th of March 2005;
  - weather forecast information can be used together with exhibition notifications in order to plan visiting an exhibition on a rainy day.

2. Arranging the trip

- Booking a flight from Munich to Lyon and back and a suitable hotel in Lyon. (These reservations are, as already explained above, updates to some Web resources that need to be executed in all-or-nothing manner.)
- Making train reservations and corresponding hotel reservations in order to visit also Orange, Arles, Nîmes, and Marseilles.

Adapting Plans to Changes

The *adapting plans to changes* subtask of organising travels consists of two phases:

1. recognise changes that might affect already made plans, and
2. react to these changes by adapting plans.

1. Recognise Changes Affecting the Plan Events such as changes of weather conditions, delays of flights or trains, and cancellations of flights can affect the arrangements made in the initial planning phase of organising a travel. Recognising such changes in very short time is the premise for adapting the initial plan, e.g. by booking another flight. Recognising changes involves communication of change notifications and detection of situations of interest.

Consider a personal travel organiser that plans travels for its owner and has also the capability to react to changes that might influence arranged plans. (A personal travel organiser is an example of a system for travel planning and support.) There are two possible strategies for making the travel organiser aware of changes:

- pull strategy, meaning that the travel organiser periodically queries the data from remote Web sites (e.g. flights schedule) in order to determine whether simple changes of interest have occurred;
- push strategy, i.e. the Web-based information systems inform the travel organiser about changes that have taken place either locally or remote (case in which the systems have been notified about these changes).

After receiving notifications about changes that might affect already made arrangements, the personal travel organiser needs to detect situations of interest, i.e. possibly time related combinations of simple happenings that might have an impact on already made plans.

2. React to Changes Of great importance in developing travel planning and support systems is the capability to automatically react to changes (to situations of interest). Based on the events that have occurred, on the detected situations of interest, actions need to be executed in order to adapt plans to these changes. Example of actions are reordering travel destinations, notifying affected friends and colleagues, finding and booking other flights, or booking overnight stays.

Scenario Consider again the scenario previously introduced; after carrying out the presented tasks, the necessary arrangements for a vacation's initial plan have been realised.

Mrs. Smith's travel organiser has the capability to detect changes that might affect the initial plan. Examples of such changes are:

(a.) The flight booked for Mrs. Smith from Lyon to Munich on 20th of March 2005 has a delay. In such a case, the new arrival time and Mrs. Smith appointments need to be taken into consideration, before deciding how to react to such a happening.

(b.) The flight booked for Mrs. Smith from Lyon to Munich on 20th of March 2005 has been cancelled. In such a case, there are two possibilities:

(b.1) the airline provides an accommodation for the night of 20th to 21st of March 2005, or

(b.2) the airline does not provide such an accommodation.

The possibilities require different reactions. In the case of flight cancellation, the travel organiser could wait for a notification regarding accommodation only a fixed amount of time (e.g. it waits 2 hours from the reception time of the notification regarding the flight cancellation).

(c.) A train, for which Mrs. Smith has a reservation, has a delay.

For detecting changes like the ones presented above, the travel organiser needs to have the capability to

- detect notifications;

- detect (possibly) time-related combinations of notifications. For example, notification regarding (b.) followed by a notification regarding (b.2).

- detect combinations of notifications that have been received in a specified time interval (note that such a time interval could also be given as a duration, e.g. 2 hours);

- compare time notions (e.g. in case of a notification regarding (a.), the arrival time should be *before* 5:00h on 21st of March 2005).

The scenarios introduced here are, of course, simplified. A corresponding real life application would probably require more details, more features, and therefore more complex reactions. Nonetheless, the use case *Travel organisation* stresses salient aspects of reactivity on the Web. The scenarios are specific to a class of applications. Arguably, many other different application classes, such as Web service deployment, Web-based workflows, e-commerce, e-learning, share "reactivity traits" with the scenarios presented here.

1.3 Concepts

This section informally introduces the concepts that are needed for discussing the thesis' solution for reactivity on the Web. More formal definitions of the concepts can be found in Chapter 4.

For realising systems having the ability to accomplish tasks like the ones presented in the application scenarios of Section 1.2.1, the following are required:

- querying Web resources' data and reasoning with these data, e.g. for searching for the cheapest suitable hotel;
- communicating notifications between systems that (in some sense) cooperate to realise tasks;
- querying and reasoning with notifications' data;
- updates to (local and remote) Web resources;
- support for executing updates in an all-or-nothing manner;
- detecting situations of interest;
- reaction capabilities, e.g. to look for and book another flight as reaction to flight cancellation.

The above mentioned requirements are not tailored to the application scenarios, examples of reactive Web applications, that have been introduced in the previous sections. Instead, these are general requirements that need to be met when proposing solutions for realising reactivity on the Web. Other application scenarios have also been explored but no other requirements have been recognised.

1.3.1 Events

Informally, an *event* is a happening to which each Web site (through a reactive program) may decide to react in a particular way or not to react to at all. For example, an update to a Web resource, a query posed to Web resources, or just "8 o'clock every morning" are events.

One might argue that defining an event in such a way is too vague. But, the intention here is to emphasise that one can conceive every kind of changes on the Web as events. However, each Web-based reactive system can be interested in different types of events or in different combinations of (like a given temporal order between) such events. Thus, the large spectrum of possible events is always filtered relatively to one's interests (e.g. the owner of a personal travel organiser). Thus, the events a Web-based reactive system is interested in are in a sense subjective.

A *situation* is a combination (like conjunctions or disjunctions) of more than one events. For example, a sequence of events that have occurred in a specified time interval is a situation. A concrete example of such a situation has been given in Section 1.2.1, consisting in a flight cancellation notification followed by a notification saying that the airline does not provide an accommodation.

1.3.2 Communication Strategies: Push and Pull

The need and importance of making Web sites aware of events that have occurred on the Web, through notifications (i.e. data about events) that are communicated between Web sites, has been made clear through the motivating application scenarios. As exemplified in Section 1.2.1, two strategies are possible for communicating notifications:

- a *push* strategy, where a Web site informs possibly interested Web sites about events, and
- a *pull* strategy, where interested Web sites query periodically (poll) data found at other Web sites in order to determine changes.

Both strategies are useful. The pull strategy is supported by Web query languages like XQuery [158] or Xcerpt [53]. For propagating events (i.e. communicating data about events), a push strategy has several advantages over a strategy of periodical polling: It allows faster reaction to events, as a notification is communicated as soon as possible as opposed to a detection at the next periodical pull. It saves resources, both locally and on the network. Locally, a client interested in some change of Web data does not have to store the old Web page to detect differences (changes) from the new version. On the network, a push strategy can reduce network traffic, since communication only takes place when a change has happened, and only the changes in information have to be communicated.

1.3.3 Volatile vs. Persistent Data

For detecting situations of interest, incoming events need to be queried. In order to determine what abilities such queries need to have and thus to determine if existing Web query languages can also be used for querying events, the differences between Web resources and incoming events (if they exist) need to be revealed.

Data of incoming events and data of Web resources are indeed different. To better understand the differences one should imagine data of incoming events like *speech* and data of Web resources like *written text*.

(a.) Speech is like a stream of words or propositions (some phonology works refer to the notion of *stream of speech* [94]). Considering two particular moments in time, for example during a researcher's presentation, the information "received" through speech differ in the sense that at the most recent moment one has more information as before. Still, one can not predict what information he/she will receive by the end of the presentation. Likewise, events are received in a stream-like manner. If one is interested in a particular sequence of events and such a sequence has not been entirely received until the present moment, one needs to wait to see whether other events of interest will also occur. The article the researcher is explaining in its presentation is written text, thus reading it for selecting (querying) information of interest would give the same information independent from the time point of reading.

(b.) Speech cannot be modified. If one has communicated some information in this way one can correct, complete, or invalidate what one has told – through further speech. In contrast, written text can be updated in the usual sense. Likewise, data of incoming events is *not* updatable, but data of Web resources is updatable. To inform about, correct, complete, or invalidate former events, new notifications are communicated between Web sites.

Thus, an important distinction is made between *volatile data* (data of events) and *persistent data* (data of Web resources). This distinction not only reflects the differences noted above, but, through the analogy with speech and written text, it is intended to lighten the understanding of and thus programming with a reactive language for the Web.

1.3.4 Event Queries

To query persistent data, *Web queries* are used (expressed using a query language, such as XQuery [158] or Xcerpt [126]). There are a couple of arguments showing that Web query languages are not suitable for querying volatile data.

- Query languages can be used for detecting the occurrence of single events, combinations of events that have (possibly) occurred during the time interval determined by other events, can not be easily expressed.

- Making all incoming events persistent would be a premise for using Web query languages for "detecting" situations of interest. However,

  - This is not a realistic approach to reactivity on the Web, as the amount of incoming events could be tremendous and thus very expensive regarding system resources. The application scenarios presented in Section 1.2.1 have shown that an approach to reactivity on the Web is desired that can be used by almost every person and on "small" computer systems like a personal travel organiser.

  - Detecting complex situations that are of interest is not an easy task when working only with a query language. For example, time related combinations of situations require reasoning with temporal information (based on raising or reception time of events) and combining event data. A simple event language capable of detecting situations of interest is needed in order to ease the use of it by novice practitioners.

  - The reaction to incoming events of interest would be slower as in the case of querying directly volatile data.

- Streamed implementation of query languages are not what is really needed for querying events. The streamed evaluation is intended for querying large amount of data that may be unbounded in depth and breath, i.e. for querying data streams. However, this thesis considers the scenario where a large number of events are received by a Web site, but the size of an event's data is not too big.

Thus, another kind of queries are needed for querying volatile data in order to detect situations of interest. Such queries against volatile data are called *event queries* in this thesis. Web queries and event queries can be very similar. However, event queries are more likely to refer to event sequences or time, as many Web-based applications require reasoning about events that have a temporal extent.

An event query may be *atomic* or *composite*. An *atomic event query* refers to one single event, it represents a pattern for the single incoming event that is of interest. A *composite event query* refers to one or more incoming events and queries for combinations of these, imposing also temporal restrictions on the constituent events, e.g. only events that have occurred in a specified time interval are of interest. Such a time interval can also be given by the occurrence time of other events. *Note* that composite event queries refer to an event when specifying its *exclusion* (they specify event negation). Composite event queries are intended to detect situations, cf. Section 1.3.1.

1.3.5 Raising Events

Events occur on the Web at different Web sites. From the perspective of a particular Web site, events that occur at this Web sites are local and events that occur at other Web sites (remote Web sites) are remote. Without communicating events' data between Web sites, a Web site can not be made aware of events that have occurred at remote Web sites. Thus, there is a need for propagating events on the Web in order to be able to react to events of interest that do not necessarily happen locally at a Web site.

For propagating events on the Web, Web sites need to have the ability

- to detect events that have occurred either locally or have been received as notifications from other Web sites,

- to raise notifications containing data about these events as reaction to them, and

- to send notifications to Web sites that are (possibly) interested in these notifications.

Raising events calls for constructing new notifications containing data of events that have occurred, in the sense that the data can be restructured and new data (like the URI [145] or IRI [89] address of the Web sites to be sent to, i.e. a recipient Web site or more) can be added. If no recipient Web site is added, the notification is to be sent to the Web site raising it.

1.3.6 Updates and Transactions

Updates are changes to Web resources, like insertions, deletions or replacements of parts of information found on the Web. The work presented in this thesis considers data of Web resources in XML (eXtensible Markup Language) [159] format. Thus, when discussing about updates to Web resources updates to (local or remote) XML documents are meant.

Updates can be *elementary* or *complex*. *Elementary updates* are insertions, deletions and/or replacements of parts of a single XML document that are specified through a pattern for the data to be updated augmented with the desired update constructs (insertion, deletion, replacement). *Complex updates* are combinations of more than one simple update that are, in general, to be performed on more than one (local or remote) XML documents. Such combinations of simple updates forming a complex update can be conjunctions or disjunctions. For example booking a flight *and* a suitable hotel represents a complex update. Two kind of conjunctions of updates are needed:

- *ordered* conjunctions, meaning that all the specified updates are to be executed and in the specified order, and

- *unordered* conjunctions, meaning that all the specified updates are to be executed, but the order of execution is not of importance.

In this thesis, an *action* denotes an elementary or complex update that is to be performed, raising and sending a notification, or a combination of updates that are to be executed and notifications that are to be raised and sent. Like for updates, (ordered or unordered) conjunctions or disjunctions of actions need to be considered.

Transactions are actions that are to be executed in an *all-or-nothing manner*. The concept of transactions is a well established and understood concept in the field of database systems. Transactions obey the ACID properties [136]:

- Atomicity: operations of transactions are executed in an all-or-nothing manner, that is they are never left "half-done" (as a consequence of this property, if an error occurs during a transaction partial effects must be undone),
- Consistency: assuming that all constraints (like conformance of an XML document to a particular schema) are true before executing a transaction, they should remain true after its execution,
- Isolation: transactions must behave as they were executed in isolation from each other (that is, execution of transactions must be equivalent to some serial order), and
- Durability: after the successful execution of a transaction, all effects must be durable (for example, after executing an update all modifications should remain in the respective XML document).

1.3.7 Reactive Rules

Reactivity presupposes the ability to automatically react to previously defined situations with specified actions. This can be specified and realised by means of reactive rules (also known in the literature as active rules [74] or event-condition-action, short ECA, rules [119, 141]). The general form of these rules is *on **event** if **condition** do **action*** that specify the desired reactive behaviour. At every occurrence of the event, the rule is triggered and the corresponding action is executed if the specified condition is satisfied. (The execution of a rule's action is referred to as *firing* of the respective rule.) In this thesis, the components of reactive rules are tailored to the context in which reactivity is to be realised, namely the Web. Thus,

- the **'event part'** specifies an event query, i.e. a query against volatile data,
- the **'condition part'** specifies a Web query, i.e. a query against persistent data, and
- the **'action part'** specifies a transaction or notifications to be raised.

The concepts introduced in this section are explained in more detail in Chapter 4. Other concepts are introduced throughout this thesis so as to complete the whole picture describing the proposal of this thesis – a novel programming language for reactivity on the Web.

1.4 Outline

The remainder of this thesis is structured as follows:

- **Chapter 2** gives a short introduction into Web technologies that are required for understanding the proposal of this thesis. After a few words on the *World Wide Web* and its *Consortium* (Section 2.1), Section 2.2 discusses Web data representation formalisms that are relevant for the work presented in this thesis. Models, protocols, and units for communicating data between applications on the Web are introduced in Section 2.3. The chapter ends with an overview on some of the established query languages for the Web (Section 2.4.1), offering a detailed introduction into Xcerpt (Section 2.4.2), as it represents the (Semantic) Web query language chosen to be embedded into the language proposed by this work.
- **Chapter 3** begins with a compact introduction into the concepts on which active databases build upon (Section 3.1) and continues with discussions based on existing concrete proposals for update languages for the (Semantic) Web (Section 3.2), and reactive languages for the (Semantic) Web (Section 3.3).

- **Chapter 4** represents the first chapter of the part dedicated to the language *XChange*, the reactive language proposed by this thesis, and discusses the language paradigms, concepts, constructs, and syntax. The chapter begins by presenting the paradigms upon which the language XChange relies (Section 4.1). Section 4.2 introduces the concept of *event* and discusses the kinds of events XChange supports. Section 4.3 introduces *event messages* as means for representing event data. Section 4.4 discusses means for specifying (classes of) events of interest that might require a reaction; it introduces *event queries* – queries against event data. Section 4.5 offers a short discussion on Web queries in XChange. Section 4.6 introduces means for updating data with XChange. Section 4.7 offers a discussion on complex updates in XChange and the concept of transactions comprising updates. The chapter ends with Section 4.8, an introduction into XChange rules that puts together the puzzle pieces discussed throughout the chapter.

- **Chapter 5** turns to defining the declarative semantics (Section 5.1) of the three parts of an XChange reactive rule: event query part (Section 5.1.1), Web query part (Section 5.1.2), and update part (Section 5.1.3). The chapter discusses also the evaluation of XChange event queries (Section 5.2.1), the underlying ideas of evaluating Web queries (Section 5.2.2), and of executing XChange updates (Section 5.2.3).

- **Chapter 6** gives (part of) the implementation in XChange of some of the use cases that have been developed for the language. Section 6.1 presents the use case *Travel organisation* whose explanation has been given in Section 1.2.1. Section 6.2 gives flavour of two XChange use cases, one aiming at showing that XChange can also be employed for Semantic Web applications and one at showing that XChange is suitable for implementing *business rules*.

- **Chapter 7** begins with explicitly stating the main contributions of the thesis to the research work on (Semantic) Web reactivity (Section 7.1). It continues with a discussion on some of the perspectives for further work in XChange (Section 7.2). Section 7.3 concludes this thesis.

Remark. Parts of the thesis that are explicitly marked as *Discussion* motivate some of the design decisions made (respond to questions like *Why not introduce other constructs in the language?* or *Why take one particular approach to modelling answers to queries against incoming events?*) or offer some ideas for further extensions of the language. Discussions can be skipped at first reading the thesis by programmers wanting to get fast to programming applications in XChange.

CHAPTER

TWO

Preliminaries

2.1 World Wide Web

Following the conviction that "in an extreme view, the world can be seen as only connections, nothing else" (*Tim Berners-Lee*, [31]), the hypertext and the Internet have been assembled by Tim Berners-Lee into the *World Wide Web* (or simply Web). The exponential growth of the Web is reflected in the difference between one single Web site in 1991 (the first Web site was built in 1990 and put online in 1991 by Tim Berners-Lee) and, according to a study made, more then 550 billion documents on the Web[1] ten years later. Thus, the Web started a new communication era that had a major impact on the speed with which people communicate and on the amount of data available for retrieving from all corners of the world.

"The Web provides a simple and universal standard for the exchange of information." [6]

In order to "lead the Web to its full potential" [31], to develop common protocols for the evolution of the Web, the *World Wide Web Consortium* (W3C) [3] has been founded in 1994 as a non-profit organisation. W3C is an international association of industrial companies, research centres, educational institutions, and other organisations. One of the activities of the consortium is to set standards for the Web area. W3C standards are called *W3C recommendations* as no standards can be *enforced* in an environment without a central authority. For example, XPath (XML Path Language) [149] is a W3C recommendation for a language for selecting parts of XML documents.

The Web can be considered as a success, but its inventor, and not only, strongly believes in a *"better"* Web where the information systems have advanced features that mirror some more abilities of the human brain. Thus, the *Semantic Web* vision was born, an endeavour aiming at enriching the existing Web with meta-data and data and meta-data processing to allow information systems to actually *reason* with the data instead of merely *rendering* it. Tim Berners-Lee, James Handler, and Ora Lassila have introduced in [32] the idea of Semantic Web as the future of the existing Web, coming with important improvements in Web's usage. The Semantic Web "will not be a new global information highway parallel to the existing World Wide Web; instead it will gradually evolve out of the existing Web" [15]. Industry, governments (e.g. the U.S. government through the DAML Project, the European Union through its Sixth Framework Programme[2]) are interested in and support research work in this direction. As a consequence of this strong interest in bringing more semantics on the Web, a variety of technologies have been proposed that are generally intended for machine usage and are not so easy to use by humans. However, of crucial importance for the further development of the Web is the lightness of technologies' usage (in particular the languages' usage).

This thesis proposes a novel reactive language for the Web; its design has been carefully developed so as to be easily understood and used also by novice practitioners. To some extent, the Web is regarded here from the data and technology perspectives as the thesis introduces a novel (abstract) view over the Web

[1]Wikipedia Web Statistics, `http://en.wikipedia.org/wiki/World_wide_web`

[2]European Union's FP6, `http://www.cordis.lu/fp6/`

data and a novel reactive technology (cf. Section 7.1). The research work builds upon existing foundations of the Web such as existing communication protocols, an overview of which is given in this chapter. An extensive discussion of the technical foundations of the Web can be found in Erik Wilde's book *Wilde's WWW - Technical Foundations of the World Wide Web* [142].

2.2 Data on the Web

This section introduces into representation formalisms for data on the Web that are relevant for the research work presented in the next sections of the thesis. This perspective is reflected also in the order in which data representations are shortly described; the section does not mirror the temporal order of representations' creation. The central point of the section is the Extensible Markup Language [159] whose design goals and features are introduced in Section 2.2.1 followed by a short introduction of some of its ancestors (Section 2.2.2) and friends (Section 2.2.3).

2.2.1 Extensible Markup Language (XML)

The *Extensible Markup Language* (XML) [159] has been developed by the W3C in 1996 with the intention to provide a platform for defining user-specific document types. Existing markup languages at that time, SGML and HTML (see Section 2.2.2), have had some deficiencies that motivated the development of XML, i.e. the inflexibility of HTML and the complexity of SGML (*note* that XML is a subset of SGML). SGML, HTML, and XML are all *markup languages* for representing Web data but they have different design goals.

"XML was designed to describe data and to focus on what data is."[3]

Markup

Markup is information about the logical structure of a document and it is mixed with the actual content of the document. The idea of using markup for structuring content was to allow humans and software to interpret a document unambiguously.

Structural information is defined in terms of *elements* that have a name (*label*) and content. The content of an XML element may consist in other XML elements, character data, or an element may have no content (the element is empty). For distinguishing between markup and content, special character(s) are used for the beginning and ending of the element markup (called *markup delimiters*). An element name together with markup delimiters is called *tag*. *Note* that element names are case sensitive in XML.

Developing applications where the data is to be stored and exchanged as XML implies in a sense creativity, as the name of XML elements are not predefined. Users define XML tags using a vocabulary specific for the application.

Example 2.1 (XML Elements)
An example of an XML element labelled `accommodation` containing other XML elements that carry information about two hotels in Paris.

```
<accommodation>
  <currency>EUR</currency>
  <city>Paris</city>
  <country>France</country>
  <hotel>
     <name>Princesse Isabelle</name>
     <category>3 stars</category>
     <price-per-room>112</price-per-room>
  </hotel>
  <hotel>
     <name>Corail</name>
```

[3]XML Tutorial, `http://www.w3schools.com/xml/`

```
        <category>2 stars</category>
        <price-per-room>65</price-per-room>
        <no-pets/>
    </hotel>
</accommodation>
```

Note that elements have different structure: `hotel` elements contain other elements (like `name` and `category` elements) called *subelements*, `name` elements contain character data (their *content*), and the `no-pets` element is an empty element (without content).

An *opening tag* is a tag that begins an element, like `<currency>` or `<hotel>` in the example given above, an *ending tag* is a tag that ends an element, like `</currency>` or `</hotel>` in the example. An opening tag has the form `<label attributes>`, where the attributes are optional (see Section 2.2.1), an ending tag the form `</label>`. Omitting some ending tags is not allowed in XML, but in HTML (see Section 2.2.2).

To some extent, *relations* or *properties* are defined in XML through the nesting structure. For example, the `currency` subelement in the example given above specifies that the prices for the hotel rooms are given in Euro. But, beware that this is not always enough for machine processing, because the nesting does not specify the semantics (meaning) of data. This is one of the reasons why new languages, such as RDF (see Section 2.2.3) are developed for realising a “more intelligent” Web.

Attributes

Sometimes it is necessary to include information about elements. As a consequence, the concept of *attributes* has been introduced in XML. Attributes are included in the opening tags of elements and are represented by key/value pairs of the form *name = ”value”*.

Example 2.2 (XML Attributes)
The following example is a slight modification of the XML fragment given in Example 2.1. The currency information is given here as an attribute of the `accommodation` element and not as its subelement.

```
<accommodation currency="EUR">
    <city>Paris</city>
    <country>France</country>
    <hotel>
        <name>Princesse Isabelle</name>
        <category>3 stars</category>
        <price-per-room>112</price-per-room>
    </hotel>
    ...
</accommodation>
```

The above given example shows also the freedom users have in describing the data needed for XML-based applications. *Note* that white space inside attributes’ values is significant.

There are also reserved attributes in XML, such as `xml:lang` for defining the language of the element’s content and `xml:space` for defining whitespace as significant.

XML Documents

An *XML document* consists of a document prologue followed by a document tree of elements, character data, and attributes. Document *tree* because the structure of an XML document is a strictly hierarchical topology (e.g. the `hotel` element must be ended before opening another `hotel` element). If this is the case and all entities referenced by the document have been properly declared (see Section 2.2.1), the XML document is called *well-formed*.

Document Prologue The *document prologue* contains information about properties that the XML document whose beginning it is has. Such properties can be the XML version, processing instructions or schema declaration (see Section 2.2.1). The *document declaration* (part of the prologue) is the first line in an XML document and defines the XML version (mandatory) and the character encoding used in the document (optional).

Example 2.3 (XML Document Declaration)
The document declaration of the example specifies that the XML version used is 1.0 and the encoding is ISO-8859-15.

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<accommodation>
  <currency>EUR</currency>
  <city>Paris</city>
  <country>France</country>
  <hotel>
    ...
  </hotel>
  ...
</accommodation>
```

Note that the root element of the XML document in the above example is called `accommodation`. Each XML document has one single root element.

One of the main strengths of XML consists in separating content from formatting. An XML document can be displayed in different ways without multiple copies of the same document. XML documents do not contain instructions for how the content is to be displayed. Such instructions are contained in *style sheets* that can be included in XML documents. The declaration of what style sheet is to be used and where this is to be found is part of the document's prologue. Style sheets can be written in languages like Cascading Style Sheets (CSS) [146] or Extensible Stylesheet Language (XSL) [154]. XSL is discussed in Section 2.4.1.

Encodings XML documents can contain "foreign" characters like the Romanian ă and â needed for writing e.g. the last name of the author of this thesis or the French ç for writing e.g. the first name of the author's supervisor. To support such kind of characters a variety of encodings can be specified in the document declaration. A list of frequently used encodings follow:

`ASCII` (American Standard for Character Information Interchange), 7-bit binary version of letters, numerals, and other symbols

`ISO-8859-1` (Latin, Western European without Euro), 8-bit

`ISO-8859-2` (Latin, Eastern European without Euro), 8-bit

`ISO-8859-15` (Latin, Western European with Euro), 8-bit

`Big5` (Traditional Chinese, Hong-Kong, and Taiwan), 2 bytes

`KOI8-R` (Cyrillic, Russian), 8-bit

`UTF-8` (Unicode), 1-4 bytes variable length

`UTF-16` (Unicode), 2 bytes.

For example, for writing the name of the thesis' author (specific romanian last name) in XML documents to be used in data interchange systems, the ISO-8859-2 (cited more formally as ISO/IEC 8859-2, or less formally as Latin-2) encoding would do. This encoding can be used to communicate information in languages like Bosnian, Croatian, Czech, Hungarian, Polish, Romanian, Slovenian, and other Eastern European languages.

XML offers more language constructs (e.g. comments or entities) which are not discussed here. For a detailed introduction into all XML constructs one can refer to [159] or [88].

XML Schema

A document type or schema for XML documents declares the element types available and constraints on their content, the occurrence of elements, or other details of the document structure. XML schemas are intended to ease the automated processing of XML documents. More reliable code for XML-based applications can be obtained if the parser checks for *structural validity* or performs also format-checking. A well-formed XML document that also has been validated against a document type definition is called a *valid* XML document. However, specifying schemas for XML documents is optional, i.e. unlike SGML, XML does not *require* a schema.

Document Type Definitions (DTD) [88] are part of the XML language and specify grammars for XML documents. To some extent a DTD can also be used as a type specification for XML documents. In order to win more freedom in specifying types (schemas) for XML documents, other XML schema formalisms have been introduced: *XML Schema* [156, 157] (a richer language than DTDs for specifying structure and having an XML syntax), *RelaxNG* [68] (an easy to understand and use grammar-like rule-based language), *Schematron* [92].

Document Type Definitions A DTD for an XML document can be defined in a separate file (external DTD) or within the document (internal DTD). External DTDs can be used for several documents and allow also an easier maintenance. In the case of an internal DTD, its declaration is introduced after the document declaration and before the first element of an XML document.

Example 2.4 (XML DTD)
A document type definition for the XML document used in the previous examples given as DTD could look like this:

```
<!DOCTYPE accommodation [
   <!ELEMENT accommodation (currency,city*,country*,hotel*)>
   <!ELEMENT currency       (#PCDATA)>
   <!ELEMENT city           (#PCDATA)>
   <!ELEMENT country        (#PCDATA)>
   <!ELEMENT hotel (name,category,price-per-room,no-pets?)>
   <!ELEMENT name           (#PCDATA)>
   <!ELEMENT category       (#PCDATA)>
   <!ELEMENT price-per-room (#PCDATA)>
   <!ELEMENT no-pets  EMPTY>
]>
```

The above given DTD specifies: The root element is labelled `accommodation` and contains subelements labelled `currency` (one single subelement, required), `city`, `country`, `hotel` (zero or many occurrences, denoted by `*`). Elements labelled `hotel` have subelements labelled `name`, `category`, and `price-per-room` (single occurrences, required), and optional subelements (denoted by `?`) labelled `no-pets`. Elements labelled `currency`, `city`, `country`, `name`, `price-per-room` have as content character data (`PCDATA`), and `no-pets` elements have no content (`EMPTY`).

XML References

XML offers different types of reference mechanisms, i.e. to reference parts of an XML document within the same document (using the ID/IDREF mechanism), and to link XML documents on the Web (using XLink [155]). XLink, a W3C recommendation for an XML linking language, is not introduced here as the research work of this thesis does not refer to or use it.

The *XML ID/IDREF* is a structural reference mechanism that supports cross references within an XML document, by means of two special types of attributes: identifiers (ID) and identifier references (IDREF). ID and IDREF types require a schema definition (given for example as DTD) that specifies the attributes to which these reference types are associated. The ID/IDREF mechanism is very simple:

- identifiers are used to uniquely identify elements (meaning that the value of identifier attributes need to be unique), and

- identifier references are used to reference an element inside an XML document having a unique identifier.

Example 2.5 (XML ID/IDREF)
The following example contains information about hotels. To specify the cities where the hotels are located, identifiers are used for `city` elements that are further referenced by `hotel` elements.

```
<?xml version="1.0" encoding="ISO-8859-15"?>

<!DOCTYPE accommodation [
   <!ELEMENT accommodation (currency,city*,country*,hotel*)>
   <!ELEMENT currency (#PCDATA)>
   <!ELEMENT city     (#PCDATA)>
   ...
   <!ATTLIST city id      ID    # REQUIRED >
   <!ATTLIST hotel incity IDREF # IMPLIED >
]>

<accommodation>
  <currency>EUR</currency>
  <city id="par-fr">Paris</city>
  <city id="orn-fr">Orange</city>
  <country>France</country>
  <hotel incity="par-fr">
     <name>Princesse Isabelle</name>
     <category>3 stars</category>
     <price-per-room>112</price-per-room>
  </hotel>
  <hotel incity="orn-fr">
     <name>Royale</name>
     <category>2 stars</category>
     <price-per-room>50</price-per-room>
  </hotel>
</accommodation>
```

XML Namespaces

Sometimes it is necessary to combine (parts of) XML documents to form other XML documents. If elements have same names in XML documents conforming to different schemas, by combining them conflict names appear. For disambiguation, XML *namespaces* [160] are used to qualify unique names in XML documents.

The idea is to use a different prefix for each schema (or DTD). Such a prefix is declared as an attribute, the namespace attribute, of the form `xmlns:namespace-prefix="namespace"`. The W3C namespace specification requires `namespace` to be specified through a Uniform Resource Identifier (URI) [145].

Example 2.6 (XML Namespaces)
The example gives a part of an XML representation of a notification that is communicated between two programs written in the XChange language. For a language processor to distinguish between language keywords (`event`) and the actual content of the notification (an XML element labelled `event`), namespaces are used (`xc` for the language XChange and `art` for the content of notification announcing an exhibition in Marseilles).

```
<xc:event xmlns:xc="http://www.xcerpt.org/xchange/"
          xmlns:art="http://www.artactif.com">
```

```
   <xc:sender>http://www.artactif.com</xc:sender>
   ...
   <art:event>
      <art:exhibition>
        <art:painter>G. Barthouil</art:painter>
        <art:location>Marseilles</art:location>
         ...
      </art:exhibition>
   </art:event>
</xc:event>
```

Default namespaces are namespaces specified for an element that precludes a prefix to be written to its child elements as the namespace is inherited by the element's children.

XML Tree Representation

An XML document induces a *tree* comprising the data of the document basically by labelling nodes *or* edges with tag names and by mirroring the element/subelement relationship through graph edges. Small differences between representations of XML documents have led to different data models for XML used for example in querying XML data (such as OEM [114] in XML-QL [73]).

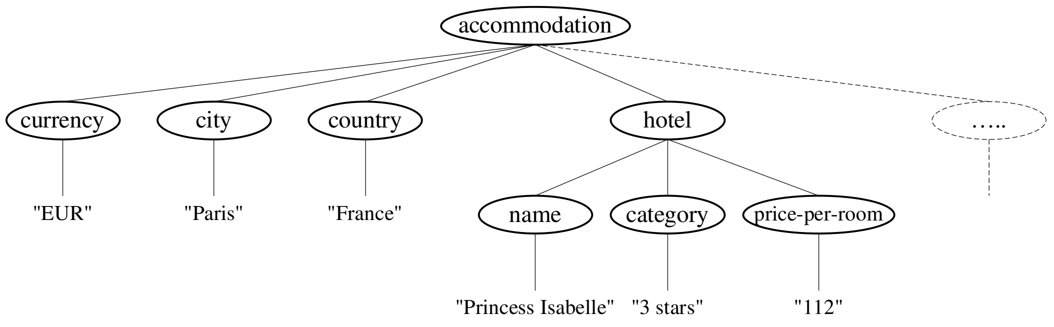
This thesis uses *node-labelled trees* for representing XML documents, i.e.

(a) element names are labels of nodes and character data content of elements are labels of leaf nodes, and

(b) element/subelement relationships are represented by edges between the nodes labelled with the respective elements' tag names.

A similar representation is used in the Document Object Model (DOM), a programming interface for HTML and XML documents. The DOM defines the way a document can be accessed and manipulated.

Example 2.7 (XML Tree Representation)
The node-labelled tree induced by the XML fragment given in Example 2.1 (note that just one `hotel` element is represented in the figure):



2.2.2 XML Ancestors

As already explained in the introductory part of the previous section, the Extensible Markup Language has been developed to overcome the shortcomings of its ancestors. This section takes a short but closer look at semistructured data, SGML, and HTML as XML ancestors.

Semistructured Data

Semistructured data [6] is explained as data without a given fixed structure or schema. In contrast, for example, data of a relational database have always a fixed schema. Semistructured data is called *self-describing* as it contains data (e.g. `Corail`) and the description of the data (e.g. `name`). This idea is encountered also in the design of the XML language.

Semistructured data can be syntactically represented by so-called *semistructured expressions*, which are expressions similar to terms in logic or functional languages. The representation is very similar to the one used throughout this thesis to exemplify the constructs of the proposed reactive language; the proposed language offers a term-based syntax as a more compact representation than an XML one.

Example 2.8 (Semistructured Expressions)
The data represented as XML in Example 2.1 is given here using semistructured expressions as representation formalism.

```
{ accommodation:
   { currency: "EUR",
     city: "Paris",
     country: "France",
     hotel:
       { name: "Princesse Isabelle",
         category: "3 stars",
         price-per-room: "112" },
     hotel:
       { name: "Corail",
         category: "2 stars",
         price-per-room: "65",
         no-pets: "" }
   }
}
```

After discussing XML and having seen the same data represented both as XML and as semistructured expressions, it is rather straightforward to understand the latter representation format. *Note* that semistructured expressions are not limited to representing tree structures, using object identifiers and references graph structures can also be represented.

Numerous representation formats for semistructured data have been developed, some of them being domain specific. OEM/Lore [114] and ACeDB [134, 52] are examples of languages for representing semistructured data. OEM, the *Object Exchange Model*, has been developed at Stanford as part of the *Tsimmis* [65] project for integrating heterogeneous data sources. (*Lore* [7] is a variant of OEM.) AceDB[4], *A c. elegans Database*, is a genome database developed primarily for storing bioinformatics data. The main challenge of the project was to deal with the flexibility of such data. As the data model used is quite general, AceDB is used also in other application areas.

Standard Generalised Markup Language

Standard Generalised Markup Language (SGML) [4] has been defined and standardised (ISO international standard 8879) in 1986. The design goal of SGML was to have "a method for describing documents in a way that makes it easy to move them from one platform to another" [56]. SGML can be considered as the base for HTML and XML, statement enforced by the fact that XML and HTML are SGML applications.

The basic idea for the development of SGML was the separation of content and presentation. The presentation lies outside the scope of SGML, the language defines only the framework for structuring content (also an abstract syntax, i.e. markup, for doing this).

In comparison to XML, SGML is more flexible and more powerful, e.g. it offers much more constructs than XML does, but at the expense of being much more expensive to implement. However, the SGML

[4]AceDB, `http://www.acedb.org`

community argue that by adding up all of the related XML standards (e.g. XLink, XML Schema) the result is not less complex than SGML. Clearly one chooses between SGML and XML depending on the requirements of its applications. A W3C note [67] of James Clark contains a detailed comparison of SGML and XML.

Hypertext Markup Language

Hypertext Markup Language (HTML) [147, 153] has been created in 1990 and has become the most popular Web standard. The reasons for its "world-wide" usage are its design goal – writing Web pages – and its simplicity. The first HTML browser (for retrieving and displaying data of Web resources) was created by Tim Berners-Lee as a researcher at CERN (Conseil Européen pour la Recherche Nucléare), Geneva.

The main use of HTML is to display information and to link documents on the Web. *How* the information is to be displayed, the *formatting* of data, is mixed with the actual information. HTML documents do not contain structural information, thus, the rendering is human-readable, but is not easy for software to "understand" the structure and content of data. Unlike XML, HTML has *predefined* tags, the set of tags is fixed and they are used to define the formatting (for example for defining lists, bold, italics, and colour).

"HTML was designed to display data and to focus on how data looks."[5]

One may think of HTML as a markup language defining only one document class. In contrast, SGML is more general and introduces the concept of document classes to describe different "classes" to which SGML documents belong to.

Example 2.9 (HTML)
The example contains the information of Example 2.1 but instead of the description of data found in XML (using XML tags) the formatting of data is given. The information about hotels are displayed in a table (introduced by the tag `<table>`), a hotel per row (introduced by the tag `<tr>`), and the name of the hotels are displayed in bold (using the tag `<b>`).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
  "http://www.w3.org/TR/REC-html40/strict.dtd">
<html>
 <head>
  <title>Accommodation information</title>
 </head>
 <body>
  <h1>Hotels in <i>Paris, France</i></h1>
  <table>
   <tbody>
   <tr>
     <td>Name</td>
     <td>Category</td>
     <td>Price</td>
     <td>Observations</td>
   </tr>
   <tr>
     <td>Hotel <b>Princesse Isabelle</b></td>
     <td>Category 3 stars</td>
     <td>Price per room 112 Euro</td>
   </tr>
   <tr>
     <td>Hotel <b>Corail</b></td>
     <td>Category 2 stars</td>
```

[5]XML Tutorial, `http://www.w3schools.com/xml/`

```
      <td>Price per room 65 Euro</td>
      <td>Pets are not allowed!</td>
   </tr>
  </tbody>
  </table>
 </body>
</html>
```

In contrast to XML, HTML permits some ending tags to be omitted in cases where the formatting is clear even with missing tags. For example, in the following HTML fragment

```
  <p>Many pages on the Web are written in so-called "bad" HTML.  <p>Omitting
  some tags is considered to be one of the reasons for this.
```

the tags denoting the ending of the paragraphs have been omitted, but HTML browsers do understand the "intended" formatting and two paragraphs are displayed. Also, HTML permits improperly nested elements within each other, like

```
<b><i>Corail</b></i>
```

and displaying ***Corail***, i.e. in bold and italics.

As not all information systems (e.g. handhelds) have the ability to interpret such "bad" markup, the *Extensible Hypertext Markup Language* (XHTML) [153] was born with the aim to replace HTML. Being a combination of HTML 4.01 elements and XML syntax (any valid XHTML document is also a valid XML document), XHTML is a "stricter and cleaner version of HTML"[6].

2.2.3 XML Friends

XML does not fulfil all requirements for a meta-data framework needed for making the meaning of data more sensed on the Web. Still, XML is a necessary part for the solutions towards the Semantic Web vision as it offers an exchange format for data *and* meta-data on the Web.

This section provides a brief introduction to the Resource Description Framework (RDF), as one of the XML friends, which provides a model for meta-data and a syntax for using and interchanging meta-data by Web-based applications of all kinds. On top of the data layer provided by the data model of RDF, a schema layer can be defined by using RDF Schema, but the need for a logical layer, i.e. for formal semantics and reasoning support, has been recognised (e.g. in [40]). Thus, the Web Ontology Language (OWL) [161] has been developed to fill the logical layer of the Semantic Web (layered) architecture. This thesis does not place emphasis on reasoning with meta-data on the Web and thus the Web Ontology Language is not introduced here.

Resource Description Framework

Resource Description Framework (RDF) [148, 15] is a W3C recommendation for a language for representing meta-data about Web resources. The concept of Web resource is more general than the one introduced in Section 1.1 and used in the thesis' work, a resource in RDF is *something identifiable* in the Web context. *Note* that this does not imply that the resource is directly retrievable, for example the thesis' author is a *person* that can be identified in RDF documents by a fragment of its home page, `http://www.pms.ifi.lmu.de/mitarbeiter/patranjan/#About_me`.

RDF statements RDF provides a framework for expressing *statements* about Web resources, i.e. to express *properties* and their *values* that resources have.

Resources are identified by Uniform Resource Identifiers (URIs) [145], more precisely by URI references, i.e. URIs together with an optional fragment identifier at the end (in the example given above,

[6]XHTML Tutorial, `http://www.w3schools.com/xhtml/`

About_me is the fragment identifier). For example, a Web page, a part of a Web page, a Web site, or a book, a car are resources.

Properties describe relations between resources, but they are also resources and thus identifiable through URIs. A Web page can have, for example, a property called `creator` whose value indicates who is the author of the page, or a property called `date` whose value indicates the creation date or the date of last modifications.

Statements are (resource, property, value) triples, where a value can be a resource or a literal (string). *Note* that RDF properties are only binary predicates that relate subjects to objects (each RDF triple is of the form *(subject, predicate, object)*).

Example 2.10 (RDF Statements)
The following example contains two RDF triples: the first one states that the organiser of the thesis' author has reserved for its owner the hotel identified by `http://www.hotel-princesse-isabelle.fr`, whose phone number is given by the second triple.

```
(http://www.pms.ifi.lmu.de/mitarbeiter/patranjan/#About_me,
http://www.myorganiser.de/has-reserved,
http://www.hotel-princesse-isabelle.fr)

(http://www.hotel-princesse-isabelle.fr,
http://www.hotel-service.org/has-phone,
"00 33 123 456 789"^^http://www.w3.org/2001/XMLSchema#integer)
```

Note the usage of a typed literal to denote that the phone number of the hotel is to be interpreted as an integer. RDF data types are based on the XML Schema framework for defining data types [157].

RDF Graph Model An RDF expression is a collection of `(subject, predicate, object)` triples that induces a graph, as each triple is represented by two nodes (labelled `subject` and `object`, respectively) connected by an arc labelled `predicate` directed towards the object node. The graph representation of RDF expressions is clearly more easy to read by humans than the XML serialisation.

XML Serialisation of RDF One possible serial representation of graphs induced by RDF statements is XML-based. The XML-based syntax is not human-friendly, but the intended "audience" are the machines as it offers a machine-processable way to represent RDF statements. At present, the arena of possible XML-based serialisations for RDF statements is rather crowded, RDF/XML [163] (W3C recommendation), Unstriped Syntax [30], RxML [131] being a few examples of existing serialisations.

Example 2.11 (RDF/XML)
The RDF statements of Example 2.10 are serialised in XML and given in the following using RDF/XML.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:organise="http://www.myorganiser.de/"
         xmlns:hotel="http://www.hotel-service.org">

   <rdf:Description
          rdf:about="http://www.pms.ifi.lmu.de/mitarbeiter/patranjan/#About_me">
      <organise:has-reserved rdf:resource="http://www.hotel-princesse-isabelle.fr"/>
   </rdf:Description>
   <rdf:Description rdf:about="http://www.hotel-princesse-isabelle.fr">
      <hotel:has-phone
        rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">00 33 123 456 789
      <hotel:has-phone/>
   </rdf:Description>
</rdf:RDF>
```

RDF Schema RDF Schema (RDFS) [162] provides extensions to RDF for defining the *vocabulary* of and constraints on data of RDF statements. This does not imply that using RDFS *one* vocabulary per domain is to be defined and used on the Web. Lacking of a central authority, the Web gives its users the freedom to define their own RDF vocabularies if this is considered useful. However, only the best (or perhaps best-marketed) ones will be used by communities and thus survive.

RDFS provides a type system for RDF supporting the description of application specific *classes* and *properties* and the specification of which properties are used within the classes. *Note* that, in contrast to XML Schema, RDF Schema does not impose syntactical constraints on the data, instead it provides information about how the data is to be interpreted. Thus, through RDFS semantic information can be made machine-accessible [15].

2.3 Communicating Data on the Web

An important issue in Web application development is to support Internet-based communication of data between different applications distributed over the network. Communication of data follows a model suitable for the application domain and it is based on network protocols (that specify a set of rules for a particular type of communication). Moreover, existing standards for specifying the communication units ensure their understanding by applications running on different platforms.

This section introduces concrete technologies that implement the issues discussed above, namely the *peer-to-peer model*, the *Hypertext Transfer Protocol*, and *XML exchange units*. The choice of describing these is motivated by their use or important role in developing the solution to reactivity proposed in this thesis.

2.3.1 Peer-to-Peer Model

Peer-to-Peer Technology

Peer-to-peer (P2P) is an emerging paradigm that might have a great impact on distributed architectures (e.g. the Internet). The peer-to-peer model is the model of a network architecture in which participating nodes, termed peers, have the same capabilities and responsibilities. Some characteristics of peer-to-peer infrastructures:

- No central coordination, i.e. there is no central instance that coordinates the actions to be taken by peers (consequence of the fact that all peers have the same capabilities).

- Peers are autonomous, i.e. no peer can decide for other peers e.g. what actions should they execute, or what data should they make available for others. E.g. if a peer `A` sends a transaction request to a peer `B`, the peer `B` may decide to execute the transaction or not to do this voluntary work for `A`.

- No peer has global view of the system. However, a peer can retrieve some information from other peers by *querying* the data of these peers, but only if the data have been made available for others. *Note* that no peer has the capability to *modify* the data of other peer.

- Global behaviour is the result of local actions. Local actions are executed by applying local decisions or as requests from other peers. The peers of a network do not have a single, common goal for which they *work* as a group, thus one can not speak about global actions.

A kind of mixture of centralised and decentralised model is also possible within a peer-to-peer model. Such a model is termed as a *hierarchical model* where special peers, so-called super peers, are used to store information about a group of peers and to do a lightweight coordination of the group. The hierarchical model has been used e.g. in [115], work described shortly in Section 3.3.

The peer-to-peer technology provides support for *direct exchange* of data and services between peers. Thus, the need of centralised servers is eliminated.

Peer-to-peer communication

Peer-to-peer communication is a communication model used in decentralised environments where each party (i.e. peer) has the same capabilities and each party can initiate a communication session.

An example of a model that contrasts the peer-to-peer model is the client/server model. In the client/server model requests and answers to these requests are communicated as in the following scenario:

- clients send requests to a central server, and
- server sends answers to requests to the clients.

In contrast, in the peer-to-peer model peers have the capability to send requests to other peers, and also the capability to answer requests of other peers. Thus, a peer has to fulfill the role of a client *and* that of a server. The client/server model might be, to some extent, seen as a special case of the peer-to-peer model.

2.3.2 Communication Protocol

Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) [142] is the Web's protocol for transferring information and, hence, is implemented by all browsers. Other protocols, e.g. File Transfer Protocol (FTP) [122], are in some cases also integrated into the browser.

HTTP is a request/response protocol based on a connection-oriented transport service. It uses two basic roles: *client* (basically sending requests) and *server* (sending responses). In addition, intermediaries (*proxies*) may be used in the request/response chain.

The design goals of the first version of the HTTP protocol (now referred to as HTTP 0.9) have been: light protocol (i.e. easy to implement servers and clients) and fast protocol (i.e. to facilitate fast retrieval of information). Using HTTP 0.9 clients could only request and retrieve data from servers, but could not send data to servers. Moreover, only text documents could be transmitted.

HTTP 1.0 [99] has been developed to overcome the limitations of HTTP 0.9. In this sense, HTTP 1.0 provides support for different media types, information about the transferred entity could be transmitted, and, by supporting new methods, clients could transmit data to servers. However, HTTP 1.0 used the same communication model as HTTP 0.9: the client establishes a TCP connection to the server, issues a request and reads back the server's response, and then the connection is closed. This led to serious performance problems (because of the use of one TCP connection per file).

HTTP 1.1 [100], the latest version of HTTP, brings improvements over HTTP 1.0 like persistent connections, i.e. after a request/response interaction the connection between the client and the server is not closed, instead the server waits for other requests from the respective client. If a client or a server does not want to keep the connection open after a request/response interaction, this option can still be specified.

HTTP Extensions

Extensions to the HTTP protocol have been proposed in order to enhance the communication with features that are desirable or even required for Web-based applications. In the following, the Secure HTTP, the HTTP Next Generation, and the Protocol Extension Protocol are shortly discussed.

Secure HTTP Secure HTTP is a secure message-oriented protocol designed to coexist with HTTP. The protocol provides a variety of security mechanisms to HTTP clients and servers, i.e. includes encryption, user authentication, and certification. These mechanisms can also be combined. Moreover, negotiation can be used in order to allow clients and servers to agree on e.g. cryptographic algorithms or certificate selection.

HTTP Next Generation HTTP Next Generation[7] (HTTP-NG) preceded actually the development of HTTP 1.1 as it supposed to be an enhanced replacement for HTTP 1.0. Extensions that were proposed in HTTP-NG regard modularisation, scalability, and extensibility. One of the ideas was to enable the sending of many different requests over a single (persistent) connection. As these requests may be asynchronous, support for asynchronicity was recognised as a requirement.

At present, the W3C does not plan to further develop the work on HTTP-NG.

[7]HTTP-NG, `http://www.w3.org/Protocols/HTTP-NG/`

Protocol Extension Protocol Protocol Extension Protocol (PEP) describes how HTTP messages can be extended with header fields extensions and new content formats. With PEP, HTTP agents can interoperate correctly with known and unknown protocol extensions, select protocol extensions available to both sides, and query partners for specific capabilities.

PEP represents one approach for an extension mechanism for HTTP. This work might be important for the future version of HTTP, as the World Wide Web Consortium is comitted to including extensibility in HTTP 1.2.

2.3.3 XML Exchange Units

As already explained in previous sections, the goal of developing XML was to have a data format for easily exchanging information between applications. The structure and meaning of data interchanged is domain and application specific. For applications running on different platforms to cooperate (in a sense) and understand the data received, simple rules defining the messaging framework are needed. Thus, the Simple Object Access Protocol [152] has born and has become a W3C recommendation.

Simple Object Access Protocol

Simple Object Access Protocol (SOAP) [152] is a lightweight XML-based protocol for exchanging information in a distributed environment. SOAP has a one-way message exchange paradigm (i.e. a SOAP message is transmitted between nodes, from a sender to a receiver), but more complex interaction patterns (e.g. request/response or multiple *conversational* exchanges) can be created by applications.

SOAP is not committed to a single underlying protocol, nor to a particular operating system or programming language. General rules are stated for the specification of protocol bindings, i.e. the formal set of rules for exchanging SOAP messages on top of a protocol (e.g. the HTTP binding in SOAP 1.2[8]). Not being tied to an operating system or programming languages, clients and service providers can communicate and exchange information (for example by requesting a method and receiving a response to it) as long as they can formulate and understand SOAP messages. Let's take now a closer look at SOAP messages.

SOAP Messages SOAP messages are used to exchange structured and typed information between systems in a decentralised environment. A SOAP message is a well-formed XML document containing an optional XML declaration and a SOAP envelope.

The *SOAP envelope* is introduced by the tag name `Soap-Namespace:Envelope` and represents the root of the XML document. The envelope is made of an optional SOAP header and a SOAP body.

The *SOAP header* is introduced by the tag name `Soap-Namespace:Header` and contains informations about how the body of the message is to be treated. For example, a SOAP header may contain a role attribute information item saying that the message is targeted only to nodes operating in the specified role.

The *SOAP body* is introduced by the tag name `Soap-Namespace:Body` and encapsulates e.g. method requests and their parameters or responses to such requests. Parameters can have primitive types (like string or integer), but complex parameters can also be handled by the SOAP messaging protocol.

There are many SOAP packages available for different programming languages (for example, the Apache SOAP for Java). Such packages take also care of the syntax details in using SOAP messages.

Example 2.12 (SOAP Messages)
A fragment of a SOAP message containing a request for a hotel reservation is given in the following. *Note* that the method requested is named `makeHotelReservation` taking parameters like the name of the person for which the reservation is to be made or the number of nights for which an accommodation is needed.

```
<?xml version="1.0"?>
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
```

[8]SOAP HTTP binding, `http://www.w3.org/TR/2003/REC-soap12-part2-20030624/soapinhttp`

```
   <soap:Body>
      <service:makeHotelReservation
       xmlns:service="urn:HotelReservationService">
           <parameter1 xsi:type="xsd:string">Paula-Lavinia Patranjan</parameter1>
           <parameter2 xsi:type="xsd:int">5</parameter2>
           ...
      </service:makeHotelReservation>
   </soap:Body>
</soap:Envelope>
```

How SOAP messages are to be processed is specified by the SOAP *processing model*, which describes:
- rules for constructing messages,
- rules by which messages are processed when received at an intermediary node or ultimate destination node, and
- rules by which portions of the message can be inserted, deleted or modified by the actions of an intermediary node. The W3C specification of the SOAP messaging framework [152] offers a more detailed discussion on SOAP processing model.

One may argue that all the issues addressed and made possible through SOAP have had already a counterpart in the CORBA framework. Different advantages are gained by using one framework or another. Still, "where SOAP really shines is as the message protocol for web services"[9].

2.4 Querying Web Data

As already pointed out through the application scenarios of Section 1.2, gathering informations from Web resources plays an essential role in realising reactivity on the Web. Accomplishing this task presupposes the existence of means for querying Web data, i.e. general purpose programming languages or Web query languages. Being focused only on the querying of data, Web query languages offer constructs tailored to the querying ability and thus are easier to understand and use by the practitioners.

An overview of Web query languages is given next followed by short introductions to some of the established query languages (Section 2.4.1). The Web query language Xcerpt is discussed in more detail (Section 2.4.2), as it represents the query language chosen to be embedded in the solution proposed in this thesis.

2.4.1 Web Query Languages: An Overview

This section is not intended to be a survey on existing Web query languages, as the focus of the thesis is not on querying Web resources. (A survey on textual Web query languages can be found in [78].) The topic of querying data on the Web has received a considerable attention; thus, the arena of existing Web query languages is rather crowded. Taking also into consideration that data is found in different formats (such as HTML, XML, or RDF), a considerable number of special purpose query languages have been developed. Though, an ideal query language should go beyond different data representation formalisms and offer an intuitive and uniform way of querying and reasoning with Web data, regardless of their representation.

Based on the selection mechanism used for retrieving parts of Web documents, two categories of Web query languages can be recognised:

(a.) *path-based* query languages that use location path specifications to address and select portions of Web documents. Such a location path consists mainly of element names or attribute qualifiers separated by forward slashes (/), very similar to a file system path.

(b.) *pattern-based* query languages that use pattern (or template) specifications for the data to be queried. Such patterns are possibly incomplete (some parts are left out and others are represented by variables) models for Web documents. In order to retrieve data from these documents they need to exhibit the query pattern used.

[9]SOAP Basics, `http://www.soapuser.com/`

This section continues with short introductions to established Web query languages as instances of the two language classes explained above. Query languages for meta data are not really established yet, i.e. proposals exist but there isn't a large community that uses one of them. Thus, the discussion here deals only with query languages for Web data. Still, in some cases extensions exist for working also with meta data (XQuery) or the language is generic enough to deal with both kinds of data (Xcerpt).

XPath

The *XML Path Language* (XPath) [149] is a language for selecting parts of an XML document. Considering query language a language capable of querying data and transforming data or constructing new data, XPath is not a query language as it lacks construction abilities. Still, XPath offers a powerful selection mechanism upon which other query languages are built (for example, XQuery and XSL).

XPath expressions specify navigation steps that operate on the tree data model of an XML document. These navigation steps are considered relative to a context node which initially is the root of the document.

Example 2.13 (XPath Expressions)
The following XPath expression specifies that the tree representation of an XML document is to be navigated from the root which should be labelled `accommodation` to the `hotel` elements that have a subelement `price-per-room` with content less than 100. From the `hotel` elements satisfying this constraint, the subelements labelled `name` are to be retrieved (or selected).

```
/accommodation/hotel[price-per-room < 100]/name
```

Considering the above given XPath expression posed to the XML document of Example 2.1, the element

```
<name>Corail</name>
```

is selected. As easily recognisable from the given example, XPath does not have an XML syntax.

XSL

The Extensible Stylesheet Language (XSL) [154] is a formatting and transformation language used to transform XML documents in other documents, e.g. XML or HTML documents. The transformation is specified by means of XSL Transformation (XSLT) [150] rules, called templates, that are recursively applied to the nodes of a single XML document. Transformation rules use guards given by patterns to restrict the nodes to which they are to be applied to. Patterns are specified in terms of XPath expressions.

XSLT has been primarily developed as a style sheet language, but it can be used also as a query language. *Note* that XSLT is a rule-based language (having an XML syntax), but the querying and constructing are intertwined.

Example 2.14 (XSLT)
The following example uses XSLT rules (introduced by `xsl:template`) to select the names of all hotels where a room costs less than 100 Euro and return them in a `result` element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/accommodation">
  <result>
    <xsl:apply-templates/>
  </result>
</xsl:template>
<xsl:template match="hotel[price-per-room < 100]">
   <xsl:apply-templates select="name"/>
</xsl:template>
<xsl:template match="name">
```

```
  <xsl:copy-of select="."/>
<xsl:template/>
</xsl:stylesheet>
```

Three transformation rules are specified in the above given example:

1. the first rule applies for the root of the transformed document if its label is `accommodation` and constructs a `result` element that will contain the outcome of applying the example's rules;

2. the second rule applies for `hotel` elements with a `price-per-room` child having a content less than 100 (the set of nodes to which a rule is to be applied to contains the elements that match the specified XPath expression), the rule selects the `name` elements of the selected hotels;

3. the third rule applies for `name` labelled elements and selects the content of them.

XQuery

XQuery [158] is the current W3C recommendation for an XML query language. XQuery uses paths for selecting data from XML documents and patterns for constructing new data. Paths and patterns are intermixed, i.e. there is no clear separation of query and construction parts. This represents a drawback, as the structure of complex programs (that usually are needed in practice) is not so easy to grasp.

An XQuery query is a so-called FLWR (pronounced "flower") expression that returns a value, i.e. an expression of the form For-Let-Where-Order-Return that generalises the SELECT-FROM-HAVING-WHERE expressions from SQL. The query parts are shortly explained in the light of the next example.

Example 2.15 (XQuery)
The following XQuery query constructs a `result` element containing the name of all hotels found in the XML document named `hotels.xml` where the price per room costs less than 100 Euro. *Note* that the same query is specified in Example 2.14 using XSLT.

```
FOR $hotel in document("hotels.xml")//accomodation/hotel
LET $price:=$hotel/price-per-room
WHERE $price < 100
ORDER BY $price
RETURN
  <result>
    {$hotel/name}
  </result>
```

The `FOR` iterates through each of the nodes resulting from the path expression that follows it, i.e. the variable named `hotel` will be bound to each of these nodes. The `LET` picks up `price-per-room` subnodes in a variable named `price`. The value of this variable is used then in the `WHERE` and `ORDER BY` parts to restrict the set of possible candidate nodes and to order the nodes of the result, respectively. The `RETURN` clause constructs the result of the query.

XQuery has been developed for querying XML data and there is no support for querying and reasoning with data and meta data. The language needs to be extended in order to be able to work also with meta data. One such extension, MetaXQuery [93], has already been proposed for querying XML and RDF data.

2.4.2 The Web Query Language Xcerpt

Xcerpt[10] is a Web query language developed as a research project at the Institute for Informatics, University of Munich. Xcerpt has been the subject of the Ph.D. project of Dr. Sebastian Schaffert, whose Ph.D. thesis [125] written under the supervision of Prof. Dr. François Bry contains a more detailed discussion on Xcerpt. This section introduces the Web query language Xcerpt, as it is part of the reactive language presented in this thesis. There are a couple of reasons why Xcerpt and not another Web query language has been chosen: Both languages – Xcerpt and XChange, the reactive language proposed in this thesis – intend to prove that

[10] Xcerpt Project, `http://www.xcerpt.org`

a *pattern-based approach* is amenable not only for querying Web data but also for specifying reactivity on the Web. Being in the Munich group the author had the chance to directly work with Prof. Dr. François Bry and Dr. Sebastian Schaffert, the main contributors to the Xcerpt project. Last but not least, *Simulation Unification* – a novel unification method developed as part of the Xcerpt project for querying Web data – plays an essential role for querying events.

This section discusses the following: The design principles of the Web query language Xcerpt are shortly explained. The informal semantics of language constructs is given through simple examples; the underlying ideas of the declarative semantics of Xcerpt queries are postponed to Section 5.1.2. The basic ideas of Simulation Unification are also touched on. The section ends with a discussion on visual rendering of Xcerpt programs and research efforts towards verbalising Xcerpt.

Design Principles

The development of the Xcerpt language followed the design principles listed and shortly explained next. A more detailed discussion of them is to be found in [125] and a gentle one in [54, 42].

Pattern-Based Queries. In contrast to the approach taken in most of existing XML query languages (where *paths* are used for selecting data), Xcerpt uses *patterns* for both selecting data items *and* constructing new data. An Xcerpt pattern is like a *form* that gives an *example* of the data that is to be queried or to be constructed, like query atoms in logic programming.

Incomplete Patterns. As data on the Web (see Section 2.2) have different and irregular structure and a schema is not always accessible, a query language for the Web needs to be tailored to these features. Thus, Xcerpt has the ability to specify incompleteness in *breadth* (not all subelements of an element need to be specified) as well as in *depth* (the path between elements in the data tree is not completely specified). Also, Xcerpt supports *ordered* and *unordered* pattern specifications to express that the order of subelements is of importance or not.

Rules. Xcerpt is a rule-based language, i.e. an Xcerpt program consists of deduction rules that may interact via (possibly recursive) rule chaining. Rules are a suitable and easy to comprehend mechanism for inferring new data from existing data (data of Web resources and views' data), very much like rules in logic programming.

Backward Chaining. In the Web context, a data driven approach to rules' evaluation is not suitable as one might need to consider as the initial point for evaluation the whole Web. In contrast to such an approach, known as *forward chaining*, a goal driven evaluation of rules is needed where only data of such resources are retrieved that are necessary to answer the query. Such an approach is known as *backward chaining* and is used in Xcerpt.

Separation of Querying and Construction. One of the drawbacks of other existing query languages for the Web (such as XQuery or XSLT) is that the query and construction parts are mixed and thus the overall structure of complex programs is not evident anymore. Thus, one of Xcerpt's design principles is the strict separation of querying and construction.

Reasoning Capabilities. Xcerpt has been primarily developed to query and reason with XML data, but due to its generality data and meta-data (for example given in RDF) can be queried in an uniform manner. Moreover, special reasoning capabilities (e.g. location or temporal reasoning) can be either directly implemented or plugged in (like a library that is used only when needed) the language.

Language Constructs

The language Xcerpt offers programmers the freedom to choose between two syntaxes for writing query programs, an XML syntax and a compact syntax where the building blocks are *terms*. The latter is used in this introduction to Xcerpt for readability and space reasons. Terms denote query patterns (*query terms*), construction patterns (*construct terms*), and also data items of Web contents (*data terms*). Common to all terms is that they represent tree or graph-like structures. The children of a node – *subterms* of a term – may either be *ordered*, i.e. the order of occurrence is relevant, or *unordered*, i.e. the order of occurrence is irrelevant. In the term syntax, an *ordered term specification* is denoted by square brackets [ ], an *unordered term specification* by curly braces {}.

Data Terms Data terms represent data items (e.g. HTML or XML documents) that are found on the Web. In an Xcerpt program the Web resources to be queried are specified using the keyword `resource` followed by the Web address(es) where the data is to be found.

Example 2.16 (Xcerpt Data Terms)
The fragment of an XML document containing information about hotels that has been given in Example 2.1 is given in the following using the Xcerpt term syntax.

```
accommodation {
  currency {"EUR"},
  city {"Paris"},
  country {"France"},
  hotel {
    name {"Princesse Isabelle"},
    category {"3 stars"},
    price-per-room {"112"}
   },
  hotel {
    name {"Corail"},
    category {"2 stars"},
    price-per-room {"65"},
    no-pets { }
  }
}
```

It is rather straightforward to understand the term syntax used by Xcerpt (and also for exemplifying the reactive language proposed in this thesis) after having seen the same fragment of data in its XML representation (Example 2.1) and also in its tree representation (Example 2.7).

Query Terms. Query terms are (possibly incomplete) patterns for the Web data that is to be queried and from which parts (subterms of data terms) are to be retrieved.

Total (complete) or *partial* (incomplete) query patterns can be specified. Partial query specifications are useful when the structure of the queried documents is not completely known, but also for minimising the terms that need to be written for meeting users query requests. *Single* square brackets [ ] or curly braces { } denote total specifications and *double* square brackets [[ ]] or curly braces {{ }} denote partial query specifications. The syntax mirrors semantical differences between the two query specifications that are explained in the following by examples.

Example 2.17 (Total vs. Partial Query Specifications)
The example contains two Xcerpt query terms that specify patterns matching with data terms having a root labelled `accommodation` with a subterm labelled `hotel`.

```
accommodation {                        accommodation {{
  hotel { }                              hotel {{ }}
 }                                      }}
```

The left query term is a total pattern specification that matches only data terms having a single `hotel` subterm that is empty. In contrast, the right query term is a partial pattern specification that matches data terms where the root may contain other subterms than the `hotel` subterm (note that it is possible to have more than one `hotel` subterm), and no constraints are imposed on the content of `hotel` subterms (i.e. may have empty content or arbitrary content).

Considering the above given query terms and the data term of the Example 2.16, only the query term on the right matches the data term (one of the reasons is that the root element contains also other subterms than `hotel` subterms).

Example 2.18 (Ordered vs. Unordered Query Specifications)
Two Xcerpt query terms that specify patterns matching with data terms having a root labelled `accommodation` with a subterm labelled `hotel` are given in this example too. By using square brackets instead of curly braces in the query term on the left, this query term matches only data terms with *ordered* subterms of the root. The query term on the right specifies that the order is of no importance.

```
accommodation [[                        accommodation {{
  hotel {{ }}                             hotel {{ }}
 ]]                                      }}
```

Because the data term of the Example 2.16 has unordered subterms, only the query term on the right matches the data term. *Note* that unordered subterm specifications match with unordered or ordered subterms of a data term (clearly, in the case that the labels and the structure is preserved).

Using the language constructs introduced previously, one can specify queries for determining if the specified patterns match with data terms found at different Web resources or not. Mechanisms are needed for selecting parts of Web documents and to use them in constructing new data items. Variables used in query and construction patterns serve this purpose and therefore are introduced next.

Variables in Query Terms Query terms contain *variables* for retrieving data items, i.e. for selecting subterms of queried data terms. Xcerpt variables are place holders for data, like logic programming variables are. In Xcerpt, variables are preceded by the keyword `var`.

Example 2.19 (Xcerpt Variables)
The following Xcerpt query term specifies a pattern for data terms with root labelled `accommodation` and having at least two subterms, one of which labelled `city` with character data content `Paris`.

```
accommodation {{
  city {"Paris"},
  var V
}}
```

Posing such a query against a data term would entail the variable named `V` to be bound to each subterm of the root excepting *the* given `city` subterm. Considering again the data term of Example 2.16, the variable `V` could be bound to each of the following four subterms:

```
  V ↦ currency {"EUR"}
OR
  V ↦ country {"France"}
OR
  V ↦ hotel {
       name {"Princesse Isabelle"},
       category {"3 stars"},
       price-per-room {"112"}
      }
OR
  V ↦ hotel {
       name {"Corail"},
       category {"2 stars"},
       price-per-room {"65"},
       no-pets { }
      }
```

Variable restrictions can be specified using the → construct (read *as*), which restricts the bindings of the variables to those terms that are matched by the restriction pattern.

Example 2.20 (Xcerpt Variable Restrictions)
Assume one is interested in retrieving only information about hotels in Paris. The next query term is a slight modification of the Example 2.19, where the variable `V` is restricted to `hotel` subterms.

```
accommodation {{
  city {"Paris"},
  var V -> hotel {{ }}
}}
```

Posing such a query against the data term of Example 2.16 leads to the following bindings for the variable `V` (*note* that only `hotel` subterms are possible bindings):

```
V ↦ hotel {                       OR V ↦ hotel {
     name {"Princesse Isabelle"},         name {"Corail"},
     category {"3 stars"},                category {"2 stars"},
     price-per-room {"112"}               price-per-room {"65"},
    }                                     no-pets { }
                                         }
```

Construct Descendant As already seen, partial specifications in Xcerpt query terms express incompleteness in breadth. For expressing incompleteness in depth, Xcerpt offers a special construct – the *descendant* construct (denoted by the keyword `desc`). The informal meaning of an expression `desc subterm` inside a query term is that the given subterm is to be found in the queried document(s) but its depth in the document tree is unknown. The following example explains the informal meaning of the `desc(endant)` construct, its practicability, though, is better motivated by querying documents with a more complex structure.

Example 2.21 (Xcerpt Descendant Construct)
The next query term specifies a pattern for data terms with root labelled `accommodation` having a `city` subterm (whose content is to be bound to the variable `City`) and containing at some depth `name` subterms (whose content is to be bound to the variable `Hotel`).

```
accommodation {{
  city { var City },
  desc name { var Hotel }
}}
```

Again, posing the above query against the data term of Example 2.16, the following variable bindings are obtained:

```
City ↦ "Paris" AND Hotel ↦ "Princesse Isabelle" OR
City ↦ "Paris" AND Hotel ↦ "Corail".
```

Construct Without In order to pose queries expressing that a certain subterm should not be found in the queried data term, Xcerpt supports *subterm negation* (introduced by the keyword `without`). An Xcerpt query term of the form

```
Label {{ without Subterm }}
```

matches only data terms labelled `Label` that do not have subterms matching `Subterm`.

Example 2.22 (Xcerpt Without Construct)
Assume that Mrs. Smith is interested in information about hotels in Paris where pets are allowed.

```
accommodation {{
  city {"Paris"},
  var H -> hotel {{
             without no-pets {}
            }}
}}
```

Querying the data term of Example 2.16 using the above given query term, Mrs. Smith retrieves the following data item as binding for the variable

```
 H ↦ hotel {
      name {"Princesse Isabelle"},
      category {"3 stars"},
      price-per-room {"112"}
     }
```

Construct Except The Xcerpt construct `except` represents a comfortable means for retrieving subterms of data terms and in the same time leaving some of the subterms of the variable bindings out. But, `except` does not specify subterm negation, as `without` does! The informal semantics of the construct will be clearer after an example.

Example 2.23 (Xcerpt Except Construct)
In order to understand the `except` construct, the query term of Example 2.22 is explained next but using `except` instead of `without`. Here, hotels where no pets are allowed are of interest, but this information is not to be part of the binding for the variable `H`.

```
accommodation {{
  city {"Paris"},
  var H -> hotel {{
             except no-pets {}
            }}
}}
```

As for the `without` example, the above given query term is used to query the data term of Example 2.16. The following binding is obtained:

```
H ↦ hotel {
       name {"Corail"},
       category {"2 stars"},
       price-per-room {"65"}
      }
```

Construct Optional For specifying optional patterns inside query terms, Xcerpt offers the `optional` construct. An Xcerpt query term of the form

```
Label {{ optional Subterm }}
```

matches data terms labelled `Label` that *might have* subterms matching `Subterm`.

Example 2.24 (Xcerpt Optional Construct)
The following query term specifies, by means of `optional`, that if a `county` subterm is found in the queried data term then this subterm should be bound to the variable `C`. In the case that no `county` subterms exist in a data term labelled `accommodation`, the query term still matches the data term but the variable has no binding.

```
accommodation {{
  optional var C -> county {{ }}
}}
```

The above given query term posed against the data term of Example 2.16 retrieves no binding for the variable `C`, as no `county` subterm is found.

Query terms are "matched" with data or construct terms by a non-standard unification method called *Simulation Unification* [55] dealing with partial and unordered query specifications. The underlying ideas of this unification method used in Xcerpt are explained later in this section after introducing other important language constructs.

Construct Terms Construct terms are patterns that make use of variables (the bindings of which are specified in query terms) so as to construct new data terms. Being templates for new data, incomplete specifications do not make sense and thus are not allowed in construct terms. They are similar to data terms, but augmented by *variables* playing the role of place holders for data retrieved in a query. Also, construct terms may contain *grouping constructs* for collecting *some* or *all* instances that result from different variable bindings.

Example 2.25 (Xcerpt Construct Some)
Assume that the variable `Hotel` is to be bound to subterms containing information about hotels by querying data of one or more Web resources. The following construct term is used to construct a new data term with root labelled `result` and containing two subterms (nondeterministically chosen from the bindings for `Hotel`).

```
result {
   some 2 var Hotel
  }
```

Example 2.26 (Xcerpt Construct All)
This example makes the same assumption as the previous one, namely that the variable `Hotel` is bound to subterms containing information about hotels, but here the variable `Price` is bound to the price per room of the selected hotels. The following construct term is used for constructing a new data term containing *all* bindings for the variable `Hotel` ordered ascending by the price per room.

```
result [
   all var Hotel order by ascending [var Price]
  ]
```

Xcerpt grouping constructs, in particular the `all` construct, are very useful in easily implementing real life problems where querying Web resources play an important role.

Construct-Query Rules Construct-query rules (short rules) relate a construct term (introduced by the keyword `CONSTRUCT`) to a query (introduced by the keyword `FROM`) consisting of AND and/or OR connected query terms. Queries or parts of a query may be further restricted by constraints (e.g. arithmetic constraints) in a so-called condition box (introduced by the keyword `where`). The `where` clause has been introduced to source out all restrictions that are not pattern-based and, thus, to keep patterns for the queried data as "clean" as possible.

Resource specifications, introduced by the keyword `in`, accompany Xcerpt queries to denote the Web resources that are to be queried. If no resource specification is given, the result of the rules of the program are to be queried. A resource specification may be atomic or compound. Atomic resource specifications comprise a URI and optionally a format for the data at the given URI. Compound resource specifications assert that multiple resources are queried. A compound resource specification is a combination of `and` or `or` connectives and atomic resource specifications.

Example 2.27 (Xcerpt Construct-Query Rules)
The data found at `http://www.hotels.net` and `http://www.hrs.fr` are queried for gathering information about hotels cheaper than 100 Euro. The bindings for the variables are then gathered in the construct part of the following rule so as to construct a view over hotel data.

```
CONSTRUCT
  result {
    all hotels {
           city { var City },
           list [
              all hotel {
                     name { var Name },
                     phone { var Phone },
                     price { var Price },
                     all optional var OtherInfo
              } order by ascending [ var Price ]
           ]
     }
  }
FROM
  or {
    in { resource {"http://www.hotels.fr"},
         hotels {{
             hotel {{
                city { var City},
                name { var Name },
                phone { var Phone },
                price-per-room { var Price },
                optional var OtherInfo
             }}
         }}
    },
    in { resource {"http://www.hrs.fr"},
        logement {{
            hotel {{
               ville { var {City}},
               nom { var Name },
               telephone { var Phone },
               prix { var Price },
               optional var OtherInfo
           }}
        }}
   }
 } where  var Price < 100
END
```

Note that

(a.) the `where` clause restricts the variable `Price` and thus only hotels where the price per room is less than 100 Euro (in this case) are retrieved;

(b.) by using the grouping construct `all`, the rule gathers all hotels for each city found in the data of the two Web resources;

(c.) the variable `OtherInfo` is used for gathering all existing information about hotels the two Web resources have; the `optional` construct is used because there is no knowledge whether the data at the Web resources have more than three terms or not.

A slight modification (from the syntax point of view) of construct-query rules called *goals* in Xcerpt are used to output the result of a rule in a specified document and using a given format. The difference in evaluating Xcerpt rules is that the result of construct-query rules can be further queried by other rules while the result of goals can not feed further processing.

Rule Chaining Complex querying problems can be elegantly solved by using Xcerpt: rules are means for structuring complex programs (keeping a clear overall structure of programs) and the *chaining of rules* (i.e. rules can query the result of other program rules) is the mechanism through which complex programs can be realised. Xcerpt rules can be evaluated in two directions: *Forward chaining* is data driven and is useful e.g. "when creating a static Web site (consisting of one or more Web pages) from an input document and an Xcerpt program"[126]. *Backward chaining* is goal driven and is necessary when querying large amount of data, such as the Web itself.

Example 2.28 (Xcerpt Rule Chaining)
The rule of Example 2.27 (above) constructs a list of hotels. Now assume that Mrs. Smith's secretary needs the phone numbers of hotels in Paris. By querying the result of the previous example, the following Xcerpt rule constructs a list containing the desired information.

```
CONSTRUCT
  info [
    all hotel { var Name, var Phone }
   ]
FROM
  result {{
    hotels {{
       city { "Paris" },
       desc hotel {{
             var Name → name {{ }},
             var Phone → phone {{ }}
            }}
    }}
  }}
END
```

Note that the scope of an Xcerpt variable is the entire rule which uses it. Moreover, a (same) variable used in the query terms of an Xcerpt query needs to be bound to the same data terms, i.e. Xcerpt uses *logical variables*.

A more comprehensive explanation of Xcerpt constructs accompanied by intuitive examples can be found in [126, 125].

Simulation Unification: Basic Ideas

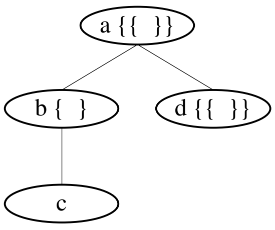
Simulation Unification [55] is a non-standard unification method developed for matching query terms with data or construct terms. It has been developed so as to cope with partial specifications, variable (not fixed) arity, and order in query terms.

Simulation Unification is based on a graph relation and tries to match the graph induced by a query term with the one induced by a data or construct term. An informal introduction to Simulation Unification is offered here by means of simple examples; for a more detailed discussion on graphs induced by Xcerpt terms see [125].

Example 2.29 (Graph Induced by Query Term)
Consider the following Xcerpt query term

```
a {{ b {c}, d {{}} }}
```

The graph induced by a query term offers another representation of the query term, a graph-based one. Thus, the induced graph contains information about the structure of the query term, the kind of specifications used (partial or total, ordered or unordered), the labels of terms, and other language constructs used. The query term given above induces the following graph:

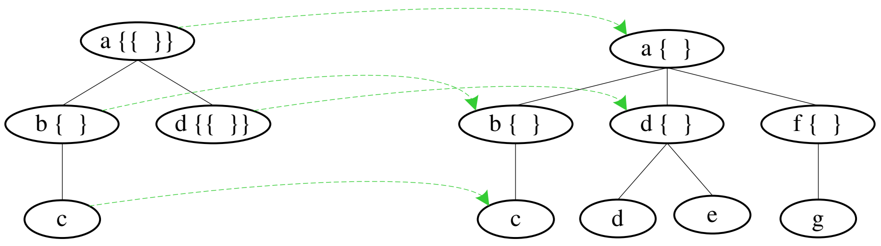


When matching, or *simulation unifying*, query terms with data or construct terms one tries to find for each subterm of the query term a “suitable”, matching subterm in the data or construct term. Considering their induced graphs, each part of the graph induced by the query term needs to have a matching part in the graph induced by the data or construct term. Clearly, the semantics of language constructs needs to be taken into account when looking for suitable matches.

Example 2.30 (Rooted Graph Simulation)
Assume that the query term of Example 2.29 is to be matched against the data term

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

The following figure illustrates the rooted graph simulations between the graphs induced by the two Xcerpt terms:



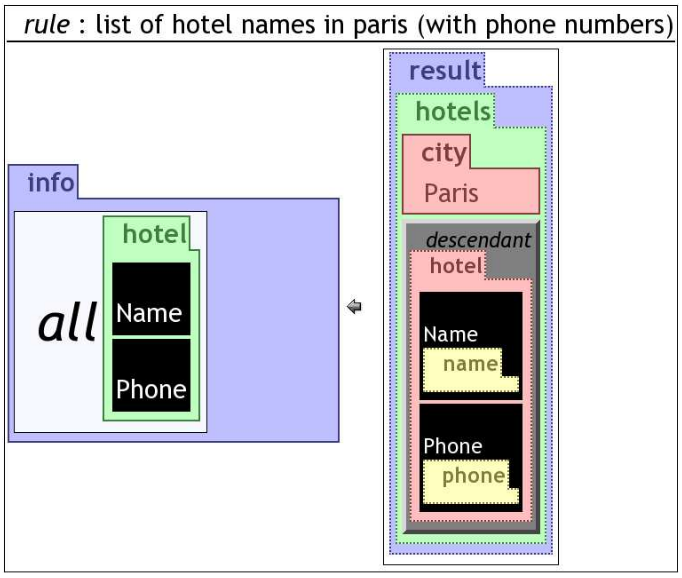
Applying Simulation Unification to a query term and a data or construct term results in a set of substitutions (for the variables occurring in the query term) called *simulation unifiers*. Simulation of query term q in a (data or construct) term t is denoted $q \preceq t$. Applying a simulation unifier to the query term yields a *ground* query term (where the variables are substituted with their bindings) that simulates into the data term.

Visual and Verbal Rendering of Xcerpt Programs

Visual and verbal renderings of programming languages play an important role not only in easing the use of languages by programmers and novice practitioners, but also in accepting and wide-spreading the languages within different kinds of communities. The need for at least one of the two rendering possibilities (visual or verbal) is accentuated for (Semantic) Web languages, as the whole idea of the Web was to be usable by anyone. The Web’s “usage” can mean more if the available technologies are easier to use, for example by non-programmers (cf. [46]).

Visual Rendering A visual rendering of Xcerpt programs called visXcerpt [23, 26, 27] has been developed in the framework of the Xcerpt project. The visual counterpart of Xcerpt is shortly introduced here, as it might represent the building block of visualising the reactive language proposed later in this thesis (this issue is touched on later, in Section 7.2, when discussing perspectives for future research). Developing visXcerpt bears evidence that pattern-based languages are very suitable for visualising their textual version and that just *rendering* the pattern specifications suffices (no fully new language as visual language is needed).

Example 2.31 (Rule Rendering in visXcerpt)
The following visXcerpt example gives the visual representation of the Xcerpt rule of Example 2.28:



Xcerpt terms are visualised as boxes where the term label is attached as a tab on top of it and nested boxes denote term-subterm relationships. The counterpart of Xcerpt's parentheses are different kind of box borders; dotted lines express partial specifications and closed lines express total specifications. Textual adornment (corresponding to keywords) accompanied by reserved colours (black, gray, and white) are used for rendering Xcerpt constructs such as *descendant*, or the grouping constructs *all* and *some*. For example, variables are visualised as black boxes with the name of the variable as white label. How the example above shows, an arrow is used to connect the query with the construct part of an Xcerpt rule, the query is visualised on the right side and the construct part on the left.

Verbal Rendering The project VOXX [25] investigates means to *verbalise* Web queries and Web data, that is to express queries and data in a controlled language, i.e. a non-ambiguous language that resembles natural language. More concrete, VOXX aims at expressing Xcerpt rules and XML data by using (an adaptation of) Attempto Controlled English (ACE) [81] developed at the University of Zurich. VOXX's goal is to bridge the formal gap between ACE vs. Xcerpt and XML by defining a common underlying formal language from which ACE, Xcerpt, and XML can be unambiguously mapped.

CHAPTER
THREE

Related Work

Chapter 1 has motivated the need for solutions to reactivity on the Web especially through the application scenarios of Section 1.2, which pose requirements that proposals for realising reactivity should fulfil. Chapter 2 has addressed the characteristics of the Web as framework of these proposals by explaining the present stage of data representation formalisms, as well as existing technologies for retrieving data on the Web (i.e. communication protocols and units, and query languages). Taking into account the recognised requirements and based on the existing technologies, this chapter demonstrates that no existing solution to reactivity on the Web do fulfils the requirements. For this, research efforts related to the topic of reactivity on the Web are introduced here by following actually the chronological evolution of reactive technologies.

The issue of automatic reaction in response to happenings of interest has its roots in the field of artificial intelligence (production systems) and active databases. In particular, the ability to react to *composite events*, i.e. (possibly time-related) combinations of event instances, has received considerable attention (cf. [74, 119, 141]). Thus, useful concepts can be "borrowed" from active databases when investigating reactivity on the Web. However, differences between (generally centralised) active databases and the Web, where a central clock and a central management are missing, necessitate new approaches. Also, complex events reflecting a user-centered (e.g. travel planning related situations) and not a system-centered view are needed on the Web. Another important issue is that the concepts used have not always clear definitions and semantics (e.g. most of the proposals do not make a difference between the notion of event and the notion of event query – the query against events), strongly motivating one of the contributions of this thesis discussed in Section 7.1 (showing that the thesis' proposal is not just an adaption of active database technology to a new medium, the Web).

As the research on (centralised) active database systems is, to some extent, saturated and has already entailed a number of commercial systems, recent research on this topic has moved its attention to other frameworks (distributed ones), such as the (Semantic) Web. One recent proposal is found in [8], where a situation monitoring system for distributed event sources is outlined. The need for taking such kind of steps forward has been motivated in Chapter 1. Thus, new languages have been proposed first just for updating data on the Web (realised by means of *update languages*) and then also for propagating and reacting to such updates (realised by means of *reactive languages*). Most of them have been primarily developed for the Web but the research community considers now finding solutions to the new problems that arise with the Semantic Web endeavour.

The chapter begins with a compact introduction into the concepts on which active databases build upon (Section 3.1) and continues with discussions based on existing concrete proposals for update languages for the (Semantic) Web (Section 3.2), and reactive languages for the (Semantic) Web (Section 3.3).

3.1 Active Database Systems

Active database systems are (relational or object-oriented) database management systems that exhibit active behaviour, i.e. they "extend their passive predecessors with rules that describe how the database should respond to events as they take place"(Norman W. Paton, in the preface of [119]). *Reactive rules* (also called event-condition-action rules) are the kind of rules used for bringing reactivity into database management systems. *Triggers* are a special kind of reactive rules where the condition part is missing (if the condition is complicated or it asks for evaluating joins, its evaluation is quite expensive).

A considerable number of active database system prototypes and event languages have been proposed (e.g. Chimera [61, 60], COMPOSE [87, 85], HiPAC [110, 124], Ode [86, 9], REACH [165, 57], SAMOS [82, 84], Snoop [64, 63], Starburst [139, 140], the SQL3 standard [2], and commercial systems supporting triggers such as Oracle[1], Informix[2], Ingres[3], InterBase[4]) that bear evidence for the intensity of work done in the past in the active database field (an annotated bibliography on active databases can be found in [91]). Thus, in order to save space and to focus on concepts that are relevant to this thesis' proposal, the section does not represent an exhaustive introduction into *existing* active database systems. Instead, this section discusses the components of reactive rules (Section 3.1.1) and issues related to the rule execution semantics (Section 3.1.2).

3.1.1 Rule Components

This section takes a closer look at the components of reactive rules used in existing active database systems from the language design point of view, as this has a great impact on the expressive power and the complexity of the active rule language. Naturally, the design decisions mirror the application domains that the developed active systems needed to address, thus no system exists that realises all the issues discussed next.

Event Part Events are happenings (e.g. changes in the database or method invocations) that are of interest for a particular application. The set of all events that a database management system is notified about forms the so-called *event history*. One common trait of existing active database systems is their ability to query events that have been received in the past, this requiring the event history to be stored completely. This approach is suitable for centralised systems with no huge amount of event occurrences, but is not amenable to distributed systems with different kinds of component systems.

Events considered in active database systems are of the following types: data modification (e.g. an insert, delete or update on a particular table of a relational database), data retrieval (e.g. a SQL select operation), temporal (i.e. specify time point(s) at which a rule is to be fired), transaction-related (e.g. beginning or ending of a transaction), and external (not defined in the event language, just registered in the system).

Events may be *atomic* (also called *primitive* or *simple* in the literature) or *composite* (also called *complex*), i.e. combinations of atomic or composite events specified by means of operators:

- *logical* operators that combine events using Boolean connectives (e.g. `and`, `or`, `not`). Not all systems support the negation of (atomic or composite) events. Some of the existing systems offer generalised versions of these logical operators, e.g. the `Any` operator in Snoop having the semantics of an *n*-ary exclusive disjunction and selection (not to be confused with the `any` operator of COMPOSE denoting the disjunction of all atomic events);
- *sequence* operator supported by most of existing active systems;
- *aperiodic* operators for expressing aperiodic events bounded by the occurrence of two other events (supported e.g. by Snoop);
- *periodic* operators for specifying events that repeat themselves within a finite amount of time (supported e.g. by Snoop and HiPAC, where the default interval is the transaction in which the events occur);

[1]Oracle, `http://www.oracle.com/database/index.html`
[2]IBM's Informix, `http://www.informix.com`
[3]Ingres, `http://www3.ca.com/Solutions/Product.asp?ID=1013`
[4]Borland's InterBase, `http://www.borland.de/interbase/`

- *selection* operators for detecting e.g. the *first* event instance in a history, the *n-th* occurrence of an event, or *every n-th* occurrence of an event (operators supported e.g. by Ode and COMPOSE);
- other operators tailored to specific applications.

Some systems, such as SQL3, consider `after` and `before` kind of events where an event specification `after Event` (or `before Event`) has the meaning of firing the rule containing it *after* (or *before*, respectively) an `Event` instance has occurred. *Note* that `before` event specifications are not realistic in a decentralised framework, such as the Web.

For specifying the semantics of composite event selection [164] when the event part is not satisfied by unique composition of events, language constructs for expressing *event consumption* are introduced. Such modes of event consumption are needed for offering a precise semantics for different kinds of applications (these modes can usually be specified by the other event constructs offered, if the event language has enough expressive power). For example, Snoop [64] uses so-called parameter contexts for this purpose and defines the *recent* (takes only the most recent occurrences of event instances into account), *chronicle* (uses the chronological order of the notified events), *continuous* (each occurrence of an event is considered as possible component candidate), and the *cumulative* context (detected composite events include all occurrences of instances of component events). [8] shows how Snoop's operators in the recent, chronicle, and continuous contexts can be expressed in Amit, a situation manager proposed recently. Composite event specifications of most systems are not easy to write and understand (partly because they lack a precise or good explanation of the constructs supported), thus when combined with event consumption specifications the behaviour of the specified rules is not so easy to grasp.

Event consumption policies are employed also for analysing expressiveness and decision problems for active database event queries (i.e. queries to events); [20] introduces a formal procedure for the specification of event consumption that is the base for comparing consumption policies. A temporal logic approach for defining the semantics of composite events is used. Results on decidability and undecidability for implication and equivalence of event queries are also given.

Conditions Conditions specified in a reactive rule are checked after an event of interest occurred and before the action's execution. Such conditions (called masks in [85]) are given by database predicates (e.g. corresponding to a *where* clause in SQL), database queries, or application procedures (that do not necessarily access the database). The condition part of a reactive rule can be omitted (in most active database systems), in which case the condition is considered *true*.

The *old* state of the database (i.e. before the modification) and the *new* state of the database (i.e. after the modification) can be used in specifying conditions of some active database systems (for example in SQL3 or HiPAC, where the default interval is a transaction). Again, this approach is adequate for centralised systems and can not be used in distributed systems of large scale.

Actions Actions are executed when events have been received that instantiate the event part, provided that the condition specified in the condition part holds. Usually, there is a mechanism for passing data from the event part to the condition and action part, and/or from the condition part to the action part, and to reference these data in the respective parts. However, tailoring the language to the application scenarios considered, most common is the passing of data only between the condition and action part (like e.g. in Chimera).

As actions, existing active database systems consider data modifications (SQL insert, delete, or update operations in relational systems, or object creation, deletion, or modification through method calls in object-oriented systems), data retrieval, data definition (e.g. for creating a new table or modifying the set of existing rules by deleting, modifying, activating or deactivating a rule, modification actions are supported e.g. by Starburst), transaction control (e.g. commit or rollback), application procedures implementing the desired action to be taken. Many active database systems support not just simple actions like the ones noted previously, but also the specification and execution of a sequence of multiple actions (e.g. Chimera supports sequence of queries, updates, and operation invocations).

3.1.2 Semantics of Rule Execution

The previous section has introduced the language constructs that can be used in specifying the desired active behaviour of an active database system. To be sure that the desired behaviour is the same as the real behaviour of a set of defined reactive rules, the *semantics of rule execution* needs to be clear. Usually this implies that a formal semantics of the active language is available. A number of alternatives exist for the rule execution semantics, possibilities determined by the taken decisions relating e.g. coupling modes, rule prioritisation. A proposal for a formalism that makes all semantical choices apparent is presented in [80].

Coupling Modes "*Coupling modes* give the rule definer fine control over how a rule is to be processed relative to the transaction that triggered the rule."[141] The following coupling modes (also found in the literature as *execution modes*) are possible: *immediate*, i.e. directly after the event has been detected, *deferred*, i.e. at the end of the triggering transaction, and *independent* (or *decoupled*, or *detached*), i.e. in an independent and separate transaction. For example, Chimera supports the immediate and deferred modes, SAMOS and REACH support all three coupling modes, but REACH implements also different variants for the independent mode.

Rule Prioritisation The occurrence of an event might determine the event part of more than one rule to be instantiated. This implies that at a particular time point a *set* of conditions and actions waits to be evaluated. An order of rules' evaluation can be determined when *priorities* for rules have been specified. For example, in SAMOS one can specify that a rule should be evaluated `before` or `after` another given rule.

Rule prioritisation has been employed also for determining sufficient conditions for termination (i.e. the execution of the rules on any initial database state does not continue indefinitely) and confluence (i.e. for any initial database state, the order of rule execution does not influence the final state of the database) of a set of active rules. Such a method has been introduced in [69], where different policies for assigning priorities to active rules are given so as to guarantee both termination and confluence of set of rules.

Evaluation Algorithms for Composite Event Detection The issue of composite event detection has received considerable attention in the field of active database systems; the most popular approaches to composite event detection are touched on shortly. Based on the representation of the partial evaluations of composite events, evaluation algorithms are based upon:

- *Petri Nets.* A partial evaluation of a composite event is represented as a Petri net. This approach is used in the active object-oriented database system SAMOS [83], where detection of composite events is based on coloured Petri-nets. Here, input places correspond to successful evaluation of parts of composite events, output places to successful evaluation of whole composite events; auxiliary places, transitions, and arcs are employed for determining relations (such as temporal dependencies) between constituent events.

- *Finite State Automata.* A partial evaluation of a composite event is represented as an automaton. Occurrence of events trigger state transitions, reaching another state corresponds to a more 'richer' partial evaluation, and reaching an end state means detection of a composite event. This approach is used for the event language of COMPOSE [87, 85].

- *Trees and Bottom-Up Flow of Events.* This is the most widely used approach for detecting composite events; it has been first employed for composite event detection in Snoop [64]. The main idea is to construct an operator tree for each composite event (or more exactly of a class of composite events) that mirrors its structure; leaf nodes are atomic events and inner nodes are composition operators. Leaf nodes are fed in with atomic events that have occurred and partial evaluations are pushed towards the root of the tree. Reaching the root node in the tree means detection of a composite event.

3.2 Update Languages

As the research community has focused its work on the development of query languages for the (Semantic) Web, the development of update languages for the (Semantic) Web has not received much attention so far. Most probably this is the reason why no update language has become a World Wide Web Consortium's recommendation until now. A W3C working draft on update facility requirements for XQuery[5] has been just recently published.

The distinction between a transformation language and an update language needs to be made here clear. Transformation languages are not update languages, transformation refers to structuring the answers to a query (i.e. newly generated data), not to modifying the structure of stored data. XSLT and Xcerpt (already discussed in Section 2.4) are examples of languages developed for querying and transforming Web data. Examples of update languages developed for the Web and special purpose tools for evolution of Semantic Web data are given in the following two subsections.

3.2.1 Update Languages for the Web

Most existing proposals for update languages for Web data expressed as XML have a common feature: path-expressions are used to select nodes within the input XML document; the selected nodes are then considered as target of the update operations. This is not surprising: an update language represents an extension of a query language with update capabilities or at least needs a mechanism for selecting parts of XML documents that are to be modified. As XPath [149] (path-oriented language for addressing parts of XML documents) and XQuery [158] (query and transformation language for XML data based on XPath) are World Wide Web Consortium's recommendations, most update proposals are built upon these standards.

This section discusses existing update languages for the Web by using the following criteria: underlying query language, update capabilities, implementation, and experimental results. The proposals that do not have a name will be referred as update language or work followed by the scientific article where they have been proposed.

Update Capabilities for XQuery

A proposal to extend XQuery [158] with update capabilities is presented in [133].

Underlying Query Language. A set of primitive update operations are proposed and then incorporated into the XML query language XQuery [158].

Update Capabilities. The underlying query language is extended with a `FOR LET WHERE UPDATE` structure for specifying updates. The `UPDATE` part contains specifications of update operations (i.e. delete, insert, rename, replace) that are to be executed in sequence. For ordered XML documents two insertion operations are considered: insertion before a child element, and insertion after a child element.

Using a `FOR WHERE` clause in the `UPDATE` part, one can specify an iterative execution of updates for nodes selected using an XPath expression. Moreover, by nesting update operations, updates can be expressed at multiple levels within the XML structure.

The above presented update extensions to XQuery are presented in [101], but with slight modifications and some extended constructs (e.g. means to specify conditional updates).

Transactions. The concept of transactions (i.e. complex updates treated as one unit and executed in an all-or-nothing manner) is not investigated.

Implementation. Alternative techniques for implementing the update operations are presented in [133] for the case when XML data is stored in a relational database (i.e. XML update statements are translated into SQL [97] statements). The authors of this work believe in the importance of having techniques for updating XML data mapped to relational data. An implementation of update capabilities for XQuery that works directly with the XQuery update statements without any use of SQL statements is under development in the Galax[6] project.

Experimental Results. This is the only work on update languages for the Web that reports on implementation performance; however, the results are for the implementation using SQL statements. Using three

[5]XQuery Update Facility Requirements, `http://www.w3.org/TR/2005/WD-xquery-update-requirements-20050211/`
[6]Galax, `http://www.cise.ufl.edu/research/mobility/`

sets of test data (i.e. synthesised data with fixed structure, synthesised data with random structure, and real life data from the DBLP [138] bibliography database) experimental results were done in order to compare the techniques proposed for the core update operations (here, insert and delete).

XUpdate

XUpdate is an update language developed by the XML:DB group[7], its latest language specification was released in year 2000 as a working draft.

Underlying Query Language. XUpdate makes use of XPath [149] expressions for selecting nodes for processing afterwards.

Update Capabilities. Simple update operations to XML documents are possible with XUpdate. This simplicity regards not the kind of operations that are supported, but the lack of capabilities to express and synchronise complex updates, i.e. to specify updates and relations between them and execute the updates conformly. An XUpdate update is represented as a well-formed XML document. The XML syntax of the language makes the programming and the understanding of complex update programs very hard.

Transactions. The concept of transactions is not investigated.

Implementation. Lexus[8] is a Java implementation of the XUpdate language, its development started at the Infozone group [90] and was continued at the XML:DB group. Another processor for XUpdate, the OXF XUpdate Engine [113] implements besides the XUpdate specification a set of extensions to the language, like iteration over a node set, and function declarations.

Experimental Results. Not available.

XPathLog

XPathLog [109, 108] is a rule-based XML data manipulation language. Its primarily goal was to offer suitable means for integrating data from multiple, heterogeneous data sources.

Underlying Query Language. XPathLog uses XPath [149] as the underlying selection mechanism and extends it with the Datalog-style variable concept.

Update Capabilities. XPathLog uses rules to specify the manipulation or integration of data from XML resources. The rule body specifies bindings for variables to parts of XML documents selected using XPath. The rule head specifies the desired update operations using the bindings gathered in the rule body. The types of update operations considered in XPathLog are insertions and replacements. There is no explicit deletion operation, design decision motivated by the intended application scenarios that rely on integrating data. Using XPathLog, one can not specify and execute complex updates, such as ordered or unordered conjunctions or disjunctions of updates.

Transactions. The concept of transactions is not investigated.

Implementation. The LoPiX [144] system, which implements the XPathLog language, is based on the Florid [137] system.

Experimental Results. Not available.

XML-RL Update Language

XML-RL Update Language [105] is an update language for XML data that incorporates some features of object oriented databases and logic programming and is thus easy to use by the practitioners.

Underlying Query Language. XML-RL Update Language extends the rule-based query language XML-RL [103, 104] with update capabilities. As a design principle of the language XML-RL, the query and construction parts are strictly separated. The rule body specifies queries to XML documents and the rule head specifies the XML data to be constructed.

Update Capabilities. Five kinds of update operations are supported by the XML-RL Update Language, i.e. `insert before`, `insert after`, `insert into`, `delete`, and `replace with`. Using the built-in position function, new elements can be inserted at a given position in the XML document (e.g. for specifying

[7]XML:DB, `http://www.xmldb.org`

[8]XUpdate Processor, `http://xmldb-org.sourceforge.net/xupdate/`

insertion of an element as first child of a given parent element). Also, complex updates at multiple levels in the document structure can be easily expressed.

However, the language does not support the specification of relations between complex updates that are to be executed synchronously. Moreover, support for propagation of updates on the Web is not provided.

Transactions. The concept of transactions is not investigated.

Implementation. An implementation is not available yet. The authors of the XML-RL project are working on an implementation of the XML-RL system, which includes the query and update capabilities.

Experimental Results. Not available, as the system is not yet implemented.

Lorel

The **Lorel** [7] query language offers simple update capabilities for semistructured data.

Underlying Query Language. Lorel is the query language from Stanford's Lore [111] semistructured database system.

Update Capabilities. The Lorel query language supports just simple updates (i.e. create and delete object names, create a new object, and modify the value of an object) of nodes into the Lore data graph (the Object Exchange Model is used). There is no explicit deletion operation for objects, instead a garbage collection approach is taken. The Lorel query language has migrated to XML data, but the update features were not ported in the process.

Transactions. The concept of transactions is not investigated.

Implementation. Lorel is implemented as the query language of the Lore prototype database management system at Stanford University [132].

Experimental Results. Not available.

3.2.2 Special Purpose Tools for Ontology Evolution

Some of the existing tools for ontology evolution and works on updates in Description Logics [16] are described next. This section is not an exhaustive survey on existing tools for ontology evolution; instead, it presents a couple of approaches followed in Semantic Web projects. Some ideas and results might prove useful when porting update languages developed for the Web to the Semantic Web.

Work Proposed in [130]

This work discusses change operations on knowledge bases having *ALC* [127] as a base description logic. In *ALC* only concepts and relations (roles) between concepts are specified. The objects (i.e. instances of specified concepts) and their relations are neglected. A more detailed description of *ALC* can be found in [127].

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

It is also discussed how the proposed system can be used to model the ontology life-cycle management, i.e. the approaches to ontology version management.

In this work, issues like ontology mapping, ontology merging are neglected but mentioned as future work. Also, multi-user management and transaction management are currently missing.

Work Proposed in [96]

A framework is proposed in [96] that integrates several sources of information about ontology evolution. The ontology change information can be represented in different formalisms. For a new version of an ontology the changes can be represented as e.g. a log of changes applied to the old version of the ontology that result in the new version of the ontology, or just as the old and new versions of the ontology. The proposed framework relates the change information that is available in different formalisms, and provides mechanisms to derive new change information from existing information gathered from different sources.

In this work, an ontology of change operations for the OWL [161] knowledge model is used as a common language for the interaction of framework tools and components. This ontology of change operations is to be found also in the OntoView [95] system, which is described later in this section. The ontology of change contains basic change operations and an extension that defines complex change operations. A number of rules and heuristics are proposed for obtaining complex change operations from a set of basic operations. The authors want to experiment with these heuristics in order to test their effectiveness and to determine the optimal values for the parameters.

OntoView

OntoView [95] is a Web-based system that provides capabilities like finding, specifying, and storing the relations between ontology versions. The system is under development and currently supports RDF-based ontology languages, such as DAML+OIL [72].

The main function of OntoView is to provide a transparent interface to versions of ontologies. Therefore, the system maintains information about relations between different versions of an ontology: the descriptive meta-data (e.g. the date of a change), the conceptual relations (i.e. the logical relations between constructs in two versions of the ontology), and the transformations between the ontology specifications (i.e. a list of change operations).

The OntoView system has been inspired by the Concurrent Versioning System (CVS) [28], which is used in software development. The implementation of the OntoView system has initially been based on CVS, but the authors want to shift to a new implementation that will be build on a solid storage system for ontologies, such as Sesame [129].

Work Proposed in [123]

An interesting proposal is found in [123], which introduces update semantics into a Description Logics system. The authors argue that this problem is strongly related to the view management problem in databases.

Two kinds of assignments are considered in this work: concept assignment (i.e. expressing properties of the form *a specified object is an instance of a concept*), and attribute value assignments. An important contribution of this proposal is the use of the transaction concept. A transaction may (and generally does) contain two parts: the first part specifies elementary updates, and the second part specifies constraints, i.e. concept assignments. (A transaction is a sequence of elementary updates that are constrained through axioms of the form `object` $\in$ `concept`.)

Two types of transactions are considered: *update transactions*, i.e. the standard transactions in databases, and *completion transactions*, i.e. transactions that “revise” the existing information and do not update the information. A revision is a situation where new information is available about a world that has not changed; thus, the constraints associated with the transaction need to be checked.

Implementation issues and optimisation proposals are also discussed in [123].

Some of the query languages proposed for RDF data provide also simple update operations. For example, **iTQL**[9] offers not only queries to RDF data, but also updates and transaction management for the Kowari Metastore, an open source database for the storage of meta-data. iTQL has a syntax similar to SQL and supports insertions and deletions of data, and commit and rollback for the transactions in the Kowari Metastore database.

3.3 Reactive Languages

Reactive languages for the (Semantic) Web have beside an update language component also an event language component (for specifying events and situations of interest) and an action language component (for

[9]iTQL, `http://www.kowari.org`

specifying the action to be executed as reaction to detected events). These language components mirror actually the components of rules in active database systems. This section discusses existing reactive languages for the Web and the Semantic Web by using the following criteria: underlying query language, underlying update capabilities, reactive capabilities, formal semantics, implementation, and experimental results. The proposals that do not have a name will be referred as work or reactive language followed by the scientific article they have been proposed in.

3.3.1 Reactive Languages for the Web

Reactive languages formerly developed for the Web support *simple* update operations on XML documents, i.e. there is no support for specifying and executing (two or more) updates in a desired order and in an *all-or-nothing* manner. Moreover, these languages have the capability to react only to single event instances and do not provide constructs for querying for complex combinations of event instances (i.e. composite events can not be detected).

There is a single proposal for composite event detection *for XML documents* [29], but the kind of composite events considered are rather simple ones and the approach is not scaled to the Web. This proposal does not represent a full reactive language for the Web (thus, the criteria used for introducing reactive languages do not apply here), but it is discussed at the end of this section as such kind of work could be extended and integrated into a reactive language.

Reactive Language Proposed in [115]

In [115] **Event-Condition-Action rule languages** are proposed for XML and RDF [148]. The ECA rule language dealing with RDF data is described in Section 3.3.2.

Underlying Query Capabilities. The ECA language for XML uses XPath and XQuery to specify events, conditions and actions.

Underlying Update Capabilities. Assumes the existence of an update language for XML.

Reactive Capabilities. The components of an ECA rule of the language are:

event part `INSERT e` or `DELETE e`, where `e` is an XPath expression that evaluates to a set of nodes. The replace operation is missing.

condition part `TRUE` or XPath expressions connected using the boolean connectives (i.e. expressing conjunction, disjunction, and negation).

action part sequence of actions, i.e. $action_1;action_2;...;action_n$ where each action represents an insertion or a deletion. For insertion operations, one can specify the position where the new elements are to be inserted using the `BELOW`, `BEFORE`, and `AFTER` constructs.

The communication between ECA components is realised through a system defined variable delta (available for use in condition and action parts) representing the set of new or deleted nodes returned by the XPath expression of the event part.

Regarding the semantics of the language, an immediate scheduling of rules that have been fired is used. Updates are not immediately executed, the inserted or deleted nodes are annotated, the triggered active rules are evaluated and at the end of the evaluation, the updates are really executed.

Distributed Environment. Just some ideas are presented for the case when ECA rules are distributed on the Web. The events and actions are considered to occur at the same local peer; the coordination of conditions' evaluation is touched on.

Formal Semantics. Not provided.

Implementation. A prototype implementation for the centralised case is outlined in [115].

Experimental Results. Not available.

The Event-Condition-Action language described above is to be found also in [21]. The authors of this paper have investigated techniques for determining triggering and activation relationships between rules. These techniques can be used for analysing and optimising the behaviour of the Event-Condition-Action rules of the language.

Active XQuery

Active XQuery [34] proposes active extensions to the query language XQuery.

Underlying Query Language. The query capabilities of the language XQuery are used in order to realise the (re)active behaviour.

Underlying Update Capabilities. Active XQuery uses the update extensions to XQuery that have been proposed in [133]. These update capabilities have been described in Section 3.2.1.

XQuery updates are seen as bulk update statements because they may involve insertion or deletion of fragments of XML documents, updates specified by means of a single update specification. Bulk update statements are transformed (i.e. expanded) into equivalent collections of simple update operations. An algorithm for update expansion is outlined.

Reactive Capabilities. As update statements are expanded, triggers are activated by update operations relative to internal portions of fragments of data. In Active XQuery, priorities for triggers may be specified in the trigger definitions. At execution time, if more than one rule is triggered at the same time, the priorities are used for choosing the trigger that is to be executed next.

Syntax and semantics of the language adapts the trigger definition and the execution model of SQL3 to the XML context. The SQL3 standard specifies a syntax and execution model for ECA rules in relational databases. Two kinds of ECA rules (or triggers) are considered: `before` triggers, which conceptually execute the condition and action before the triggering event is executed, and `after` triggers, which execute the condition and action after the triggering event is executed. Thus, Active XQuery adapts the SQL3 notions of `before` vs. `after` triggers and, moreover, the `row` vs. `statement` granularity levels to the hierarchical nature of XML data.

Distributed Environment. Distribution of triggers on the Web is not discussed.

Formal Semantics. Not provided.

Implementation. The authors of Active XQuery plan to develop a prototype if and when XQuery updates will become a World Wide Web Consortium's recommendation.

Experimental Results. Not available, as Active XQuery is not implemented yet.

Work Proposed in [36]

Event-Condition-Action rules play an essential role in the realisation of Web-based systems that require a push technology, i.e. the capability of pushing relevant information to clients, by matching new event occurences with predefined user interests. Event-Condition-Action rules are used in [36] for pushing reactive services to XML repositories.

Underlying Query Language. The query language XQuery is used.

Underlying Update Capabilities. Updating data on the Web is not supported, as the focus is just on notifying users of changes of interest that have occurred (and not on performing such changes).

Reactive Capabilities. The components of an ECA rule are:

event part simple update operations, i.e. simple insertions, deletions, or updates to XML documents.

condition part an XQuery query that is interpreted as a truth value if it returns a nonempty answer. The condition part may refer to the nodes on which the events have occurred. This is realised through the predefined variables `old` and `new` that represent the nodes on which the events have occurred with their past and current values.

action part a SOAP [152] method, but restricted to implement the call to a message delivery system that transfers information to specified recipients. It is assumed that complex parameters can be passed to the SOAP method that is invoked.

Thus, the ECA rules are used only to notify remote users and therefore, can not trigger each other. This means that the termination of execution is guaranteed, but no means are provided for updating data from resources.

Formal Semantics. Not provided.

Implementation. The main ideas that can be used to implement the proposed system are presented. Of importance here is the reuse of several current Web standards and of their implementations. The DOM Event Model [151], an XQuery engine, and a generator of SOAP calls are needed.

Experimental Results. Not available.

Work Proposed in [35]

Event-Condition-Action rules are investigated in [35] as means to realise active document management systems, i.e. XML repositories with reactive capabilities. The authors argue that such systems will constitute a natural framework for the integration of services.

Underlying Query Language. Reactive capabilities are investigated in the context of XSL [154] and of Lorel. A language extension has been designed to cover the event specification, which is absent from both languages.

Underlying Update Capabilities. Simple update extensions are proposed for XSL and Lorel, respectively. The authors discuss also the problem of detecting these changes.

Reactive Capabilities. An ECA rule consists here of an event part and a condition-action part. Events are considered insertion and deletion of elements, and insertion, deletion, and update of attribute values and of CDATA and PCDATA contents. An arbitrary change of an element's content at an arbitrary level or depth is considered a composite event.

Conditions are (XSL or Lorel) queries to XML documents, and actions consist of constructing new documents and/or modifying existing documents in the document base, and then placing them into folders, publishing them on the Web, or sending them by e-mail. Thus, in the condition-action part of rules queries and actions are mixed. The functions `old` and `new` are supported to denote the values of the node before and after, respectively, the update execution.

Formal Semantics. Not provided.

Implementation. The main guidelines of the implementation of active document systems based on XSL and Lorel, respectively, are discussed.

Experimental Results. Not available.

Active View

Event-Condition-Action rules are used in the realisation of a novel view specification language, the **Active View** language [5]. The language is used to describe views of the repository data and activities of actors that participate in an electronic commerce aplication. The Active View system, which uses the specification language, acts as an application generator and generates a Web application allowing actors to perform controlled activities and to work interactively in a distributed environment.

Underlying Query Language. At the time this work was reported, there was no standard query language for XML. The authors used instead a simple query language inspired by Lorel [7].

Underlying Update Capabilities. Updates to XML data can be realised as reactions to events, and notifications can be sent to different actors as response to changes, but the issue of updating data is not explicitly described.

Reactive Capabilities. Enhanced mechanisms for notification, access control, and logging/tracing of user activities are provided. In order to realise this, Event-Condition-Action rules are used. The components of an ECA rule:

event part (remote) method calls (e.g. switch of activity), operations like write, read, append, or detection of changes.

condition part XML queries returning a boolean value.

action part (remote) method calls, operations like the ones encountered in the event part, notifications, or traces.

Regarding the propagation of changes from the view to the repository, simple updates are supported; the incremental maintenance of the view definitions is not considered yet. The propagation of changes from the repository to the view is supported in the Active View system.

Means to customize the application and its interface, not only at server side, but also on the end-user side are supported by the system.

Formal Semantics. Not provided.

Implementation. The Active View system is developed at INRIA[10] using Java[11] and the ArdentSoftware's[12] XML repository.

Experimental Results. Not available.

WebVigiL

WebVigiL [62] is a system for monitoring changes of HTML/XML documents that are part of a Web repository and for notifying users of changes of interest.

Underlying Update Capabilities. The system does not support execution of updates to HTML/XML data; changes to documents are only monitored.

Reactive Capabilities. ECA rules are used to notify users of changes of HTML/XML documents, based on user-defined sentinels, i.e. specification of changes of interests with respect to a document. E.g. one might want to be notified of changes of links or images in a Web page. With WebVigiL, notification options can also be specified, e.g. the time point at which the sentinel is to be initiated, or (for a periodic notification) the time interval after which a new notification is to be sent.

Detection algorithms have been developed for detecting changes between two versions of a document of interest. Different change detection approaches are applied for the two types of documents (HTML and XML) the system is working on. For reducing computational time for change detection in XML documents an optimisation technique is proposed.

Different methods to present the change information to the users are investigated in this project (e.g. displaying only the changes to the document, or highlighting the differences between the new and old versions of the document).

Formal Semantics. Not provided.

Implementation. A WebVigiL prototype has been implemented.

Experimental Results. The authors are currently working on evaluating the performance of change detection algorithms.

Work Proposed in [38]

A quite recent work proposed in [38] employs triggering mechanisms for exception handling in workflow-driven Web applications; here, the notion of “exceptions” is used for representing punctual situations (i.e. events) that occur during workflow processes. The work does not propose a new reactive language; thus, the criteria used in this section are restricted here to *reactive capabilities* and *implementation.*

Reactive Capabilities. A proposal for managing exceptions is presented; it can be employed for workflow-driven Web applications where different exception handling policies are possible. The approach relies on a data model for application data and workflow, and exception meta-data.

Two classifications for exceptions are given: One classifies exceptions in behavioural (use-generated), semantical (generated by applications), and system exception. The other classifies exceptions depending on the application unit where they occur in synchronous (exceptions that occur within an activity of a process) and asynchronous (exceptions that occur at any time during the process execution).

Exceptions (events) that occur on the Web can be detected, users can be notified about them; though, combinations of exceptions can not be detected. The approach is suitable for the intended application domain, i.e. detecting exceptions inside hypertext activities that are part of workflow processes.

Implementation. The exception handling approach has been specified using the WebML[13] (Web Modelling) language; a prototype implementation exists that extends the CASE tool WebRatio[14].

[10]INRIA, `http://www.inria.fr`
[11]Java Technology, `http://java.sun.com`
[12]ArdentSoftware, `http://www.ardentsoftware.fr`
[13]WebML, `http://webml.org`
[14]WebRatio, `http://webratio.com`

Work on Composite Events Proposed in [29]

A recent work on composite event detection for XML documents has been proposed in [29] by refining the event algebra of Snoop [64, 63] and introducing a new kind of context, the hierarchical one, in order to relate e.g. insertions of new elements as sublements of a same XML element.

Reactive Capabilities. The types of events considered are insertions, deletions, and modifications of element, attribute, or text nodes in an XML document. For relating occurrences of events, *path types* are defined as restricted XPath expressions. A *path instance* is a path type that identifies a node of the tree representation of an XML document. For every path type, three primitive event types are distinguished reflecting manipulations on the given path. Thus, a *primitive event* occurs when a single node is manipulated.

Composite events are combinations (conjunctions, disjunctions, and sequences) of primitive and/or other composite events. An important issue to note here is that each composite event has a given path type restricting this way the possible situations (composite events) a user might be interested in. Moreover, one can not relate (primitive or composite) events that have occurred in XML documents distributed on the Web, as the communication of event data is not covered.

For detecting such composite events, the hierarchical position of the manipulated nodes are taken into account. Thus, the *hierarchical context* is introduced, a context in which raised composite events have only hierarchical related constituents.

Implementation. A prototype implementation of the work exists; [29] briefly describes its execution model.

3.3.2 Reactive Languages for the Semantic Web

Reactivity plays an important role in realising the Semantic Web vision. The research on Semantic Web reactivity can profit from research results on Web reactivity that need to be adapted or extended. Moreover, approaches that cope with existing and upcoming Semantic Web technologies (by gradually evolving together with these technologies) are more likely to leverage the Semantic Web endeavour. Along this line, of crucial importance for the Web and the Semantic Web is the lightness of technologies' usage (in particular the languages' usage) that should be approachable also by non-programmers.

The topic of reactivity on the Semantic Web has just begun to be investigated and, hence, leaves room for interesting research issues; some languages with reactive capabilities for the Semantic Web are touched on in the following.

RDFTL

RDFTL (RDF Triggering Language) [117, 118, 116] is an Event-Condition-Action rule language for RDF repositories that has evolved from the proposal introduced in [115], where it is shown how an Event-Condition-Action language for XML data can also be used for RDF data by making use of the XML serialisation of RDF. RDFTL operates on the RDF triples/graphs; it is a path-based language using path functions and expressions á la XPath adapted to the RDF data representation formalism.

Reactive Capabilities. The components of an ECA rule of the language:

event part insertions and deletions of resources (as instances of a given class) or of arcs specified by a triple. Instead of detecting replace operations, RDFTL has the ability to detect changes of arc targets.

condition part query consisting of conjunctions, disjunctions, and/or negations of path expressions; it represents a boolean-valued expression that can make use of a $\$delta$ variable having as set of instantiations the values that have triggered the rule.

action part sequence of one or more actions, where each action represents an insertion or deletion of a resource given by its URI, or an insertion, deletion, or update of an arc. The parts of RDFTL rules communicate just through the instantiations of variable $\$delta$.

Arguably, because of its syntax, the structure of complex RDFTL programs is not easy to grasp. Also, detection of temporal combinations of events (composite events) is not supported.

Distributed Environment. A distributed version supporting ECA rules on distributed RDF repositories is under development as part of the SeLeNe project[15]. The project investigates self e-learning networks, where such a network is a distributed repository of meta-data related to learning objects. The proposed architecture contains peers and super-peers, which coordinate a group of peers. However, the accent is on infrastructure issues and not e.g. on means for communication between the peers of the network.

Formal Semantics. Declarative semantics of language or language components is not provided. Though, the rule execution semantics of RDFTL is described. For this, a recursive function is used that takes as input an RDF graph and a sequence of updates to be performed. The function gives as output the final, updated graph (the sequence of updates is empty). The immediate coupling mode and rule prioritisation are used in RDFTL.

Implementation. The architecture of a system working on centralised systems and one on decentralised systems is proposed. Their implementation is ongoing work based on the ICS-FORTH RDFSuite repository [10].

Experimental Results. Not available yet, as the language is not fully implemented. The authors intend to experiment with the language using as testbed SeLeNe's educational meta-data.

The RDFTL event-condition-action language for RDF data is a quite recent work towards a reactive Semantic Web; however, first results on RDFTL are promising.

Algae

Algae[16] is proposed by the W3C as a general-purpose RDF query language. Beside the querying capabilities, Algae supports production rules that insert data into the RDF graph.

Reactive Capabilities. The concept of *action* is used in Algae for the directives `ask`, `assert`, and `fwrule`; they determine the use of an expression specification for querying RDF data, for insertions into the data graph, or for specifying simple ECA rules. The first directive is mandatory for an Algae processor, the other two are defined in an extension module[17]. Besides insertions of RDF data, no other update capabilities (deletions, replacements) are supported. More complex updates (e.g. ordered conjunction of updates) are not provided. The rule extension to Algae supports just production rules (as simple form of ECA rules).

Algae is based on N-triples[18] for representing and querying RDF triples; the syntax is extended with the mentioned directives and constraints' specifications (e.g. arithmetic or string based).

Formal Semantics. No formal semantics is provided.

Implementation. Algae has been implemented as part of the Annotea Project[19]; the project provides a research platform for collaborative applications based on meta-data.

Experimental Results. Such results have not been published so far.

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

[16]Algae, `http://www.w3.org/2004/05/06-Algae/`

[17]Algae Extension for Rules, `http://www.w3.org/2004/06/20-rules/`

[18]N-triples, `http://www.w3.org/TR/rdf-testcases/`

[19]Annotea Project, `http://www.w3.org/2001/Annotea/`

Part II

The Language XChange

CHAPTER

FOUR

Paradigms. Concepts. Syntax

Reactivity on the Web is the ability to detect simple and composite events that occur on the Web and respond in a timely manner. It is an emerging research issue promising useful technologies for upcoming Web systems. Many Web-based systems need to have the capability to *update data* found at (local or remote) Web resources, to *exchange information* about events (such as executed updates), and to *detect and react* not only to simple events but also to complex, real-life situations. The issue of updating data plays an important role, for example, in e-commerce systems receiving and processing buying or reservation orders. The issues of notifying, detecting, and reacting upon events of interest begin to play an increasingly important role within business strategy on the Web, and event-driven applications are being more widely deployed: terms such as zero latency enterprise, the real-time enterprise, and on-demand computing are being used to describe a vision in which events recognised anywhere within a business can immediately activate appropriate actions across the entire enterprise and beyond.

In order to meet (most of) the requirements for realising reactivity on the Web, which have been revealed by developing use cases such as the one presented in Section 1.2, the novel language *XChange* [50, 49, 43, 25, 47, 19, 18, 48, 24, 17] for programming reactive behaviour and distributed applications on the Web has been developed. XChange is a high-level programming language; its name mirrors (some of) the main features an approach to reactivity on the Web should have: *exchange* of information about events and *change* (or update) of data as reaction to (local or remote) events. The language XChange has been carefully designed so as to mirror clear design principles by the syntax of the language.

Event-Condition-Action (ECA) rules are a natural candidate to implement reactive functionality. An XChange program consists of one or more ECA rules of the form *Event Query – Web Query – Action*. They specify to execute the *Action* as an automatic response to the occurrence of a situation specified by an *Event Query*, provided the *Web Query* can be evaluated successfully. This chapter provides a detailed discussion on each of the three components of an XChange reactive rule. An XChange program runs locally at some Web site — called XChange-aware Web site. It can access and modify local and remote data (Web resources) as reaction to events. Typical events include updates of data, timer events, but can also be high-level, application-dependent events, e.g. the cancellation of a flight. Programs exhibit global behaviour by reacting to changes at (remote) Web sites. In turn, these reactions can trigger further reactions at other Web sites.

The structure of this chapter is as follows: Section 4.1 presents the paradigms upon which the language XChange is built. Section 4.2 introduces the concept of *event* and discusses the kinds of events XChange supports. Section 4.3 introduces *event messages* as means for representing event data. Section 4.4 discusses means for specifying (classes of) events of interest that might require a reaction; it introduces *event queries* – queries against event data – by discussing their essential traits, their kinds, and also the notion of answer to an event query in XChange. Section 4.5 offers a short discussion on Web queries in XChange. Section 4.6 introduces means for updating data with XChange, it discusses XChange's *update patterns* and their constructs for specifying insertions, deletions, and replacements of data. Section 4.7 offers a discussion on complex updates in XChange and the concept of transactions comprising updates. The chapter ends with Section 4.8, an introduction into XChange rules that puts together the puzzle pieces discussed throughout the chapter.

4.1 Paradigms

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

4.1.1 Event vs. Event Query

An *event* is a happening to which each Web site may decide to react in a particular way or not to react to at all (cf. Section 1.3.1). In order to notify Web sites about events and to process event data, events need to have a data representation.

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

To reiterate, events should not be confused with event queries. Events are changes in the state of the world and event queries are queries against their representation.

4.1.2 Volatile vs. Persistent Data

The development of the XChange language – its design and its implementation – reflects the novel view over the Web data that differentiates between *volatile data* (event data communicated on the Web between XChange programs) and *persistent data* (data of Web resources, such as XML or HTML documents). Section 1.3.3 has offered a detailed explanation of the traits of the two kinds of data by means of the metaphor *speech* vs. *written text*.

XChange's language design enforces this clear separation (that represents one of the contributions of this proposal, cf. Section 7.1) and entails new characteristics of event processing on the Web (discussed later in Section 5.2.1). However, in applications where a part of the volatile data received by Web sites needs to be stored for a long time or forever, the data of interest can be easily made persistent. Moreover, the language XChange is flexible enough (in terms of language design and programs' evaluation) for adapting it to other kinds of application domains as the ones intended to be primarily solved by this proposal.

4.1.3 Rule-Based Language

Reactivity can be specified and realised by means of reactive rules [74, 119, 141]. XChange is a rule-based language that uses reactive rules for specifying the desired reactive behaviour (cf. Section 1.3.7) and deductive rules for constructing views over Web resources' data.

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

XChange embeds the Web query language Xcerpt (introduced in Section 2.4.2) for expressing the *Web query* part of reactive rules and for specifying deductive rules in XChange programs. *Note* that Xcerpt is deployed also for querying single occurrences of incoming events. Xcerpt (deductive) rules allow for constructing views over (possibly heterogeneous) Web resources that can be further queried in the *Web query* part of XChange reactive rules. Not only integration and restructuring of persistent data is possible with Xcerpt, but also reasoning with persistent data (given e.g. in XML or RDF format).

Complex reactive applications can be elegantly implemented in XChange, as rules are means for structuring complex programs. (This statement is sustained by the use cases implemented in XChange and presented in Chapter 6.)

4.1.4 Pattern-Based Approach

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

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

4.1.5 Transactional Reactivity

Complex Updates XChange supports the specification and execution of *simple updates*, i.e. insertions, deletions, and replacements of persistent data items, such as XML or RDF data. *Complex updates* expressing ordered or unordered conjunctions, or disjunctions of (simple or complex) updates are also offered by XChange. Such updates are required by real applications. E.g. when booking a trip on the Web one might wish to book an early flight *and* of course the corresponding hotel reservation, *or* else a late flight *and* a shorter hotel reservation. The application scenarios of Section 1.2 have motivated the need for executing such complex updates in an *all-or-nothing manner*. Thus, XChange has a concept of transactions [136].

Transactions and ACID Properties XChange transactions obey the ACID properties, i.e. Atomicity, Consistency, Isolation, and Durability, briefly explained in Section 1.3.6. Atomicity and isolation are considered in XChange, the issues of consistency and durability for transactions are currently not investigated in the project. XChange will build on standard solutions from database systems that need to be adapted to the Web.

4.1.6 Communication Paradigms

Peer-to-Peer In XChange, the *peer-to-peer* communication model (introduced in Section 2.3) is used for communicating event data between Web sites. This means that all parties have the same capabilities and every party can initiate a communication session. Event data are directly communicated between Web sites without a centralised processing of events. XChange assumes no instance controlling (e.g. synchronising) communication on the Web.

Push Strategy For communicating (propagating) events on the Web, two strategies are possible: the *push* strategy, where a Web site informs possibly interested Web sites about events, and the *pull* strategy, where interested Web sites query periodically (poll) persistent data found at other Web sites in order to determine changes. Both strategies are useful; Section 1.3.2 has pointed out advantages that a push strategy has over a strategy of periodical polling. The pull strategy is supported by languages such as XQuery or Xcerpt that query persistent data. Therefore, so as to complement the framework, XChange offers the *push* strategy. The push strategy requires event queries to be incrementally evaluated by so-called *event managers* (cf. Section 5.2.1). In the case of XChange, this is done at every XChange-aware Web site.

Communication Protocol The language XChange is not dedicated to a particular communication protocol, instead its high-level nature allows for implementing distributed, reactive applications following different rules for communicating data. However, its goal is to realise reactivity on the Web and Semantic Web and, thus, event data is communicated over the HTTP protocol (shortly discussed in Section 2.3.2). The fact that HTTP is the protocol used for transferring information on the Web is also reflected by the prototype implementation of the language XChange.

4.1.7 Composite Events Defined through Event Queries

Composite event queries allow to recognise temporal patterns over incoming events – to recognise *composite events*. The notion of composite events has no precise definition in the literature. XChange's (occurrences of) *composite events* are defined through *composite event queries* (see Section 4.4.3) – they are *answers* to composite event queries. This is a novel way of defining composite events, but the author considers it the only intuitive one. E.g. an XChange event query can ask for occurrences of an increase of share values by more than 5 percent for the company Siemens, followed by an increase of share values for the company SAP on the stock market. An answer to such an event query contains instances of the two specified component event queries (i.e. increase of share values). Another XChange event query can ask for *all* stock market reports that have been registered between the occurrences of an increase of share values for the two mentioned companies. An answer to such an event query contains, besides the instances of the events signaling an increase for the shares of the companies, all reports registered between these two instances. The capability to query for all events having a particular pattern that have occurred between the instances of two specified event queries is one of the novelties of the language XChange.

4.1.8 Processing of Events

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

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

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

Event queries need to be in such a way that no data on any event can be kept forever in memory, i.e. the event lifespan should be *bounded*. Keeping all events in memory is not a suitable approach to reactivity on the Web; the amount of events a Web site receives might be huge, causing a continual grow in storage requirements. By language design, XChange event queries are such that volatile data remains volatile. This is consistent with the clear distinction between events as volatile data and Web resources' data as persistent data. However, making *part* of the incoming event stream persistent is application dependent (for example applications where statistics over data of incoming stream play a role). The work presented in this thesis has been developed primarily for applications like the ones presented in Section 1.2.1 but does not preclude its usage for developing applications where some of the event stream's data need to be made persistent.

4.1.9 Relationship Between Reactive and Query Languages

A working hypothesis of the XChange project is that a reactive language for the Web should build upon, or more precisely embed, a Web query language. There are two reasons for this. First, specifications of reactive behaviour often refer to actual Web contents - calling for querying Web contents. Second, reactive behaviour necessarily refers to (more or less recent) events - calling for querying events. For reasons of uniformity, it is highly desirable both for users and for system developers that the languages used for querying Web contents and querying events are as close as possible to each other. *Note*, however, that querying events calls for constructs not needed for querying Web contents. For example, the interest in a conjunction of events that occur in a given time interval can not be easily specified by means of a Web query language; on the other hand, for querying data of Web resources temporal relations between parts of the data are not needed.

The query language embedded in XChange is the Web query language Xcerpt [126, 125]. In the framework of XChange, Xcerpt's capabilities are deployed for querying and reasoning with Web resources' data, and for querying (single) occurrences of events. Xcerpt's design principles and language constructs have been introduced in Section 2.4.2. Recall that the reasons for choosing Xcerpt against other existing query languages for embedding it into XChange include the pattern-based approach followed in querying and constructing Web data, and the fact that Simulation Unification can be deployed not only for querying persistent data but also for querying single occurrences of events.

4.1.10 Language Syntax

The development of the XChange project followed the conviction that a language for the Web should have three syntaxes: a *compact human-readable* syntax, a *machine-processable* (XML) syntax, and a *visual* syntax. The compact human-readable syntax should be as compendious as possible and easy to use by programmers. The XML syntax is desirable for interchanging programs and manipulate them with XML-based tools and languages (e.g. to query an update them). A visual language "can greatly increase the accessibility of the language, in particular for non-experts"[42]. However, programmers have the freedom to choose whichever syntax they prefer.

At present, the language XChange has a compact human-readable syntax (which is a term-based syntax where a term represents a Web document, a query pattern, an event pattern, or an update pattern) and an XML syntax. The development of a visual counterpart of XChange's textual language is sought for. Along this line, the visual rendering of Xcerpt programs – visXcerpt [27] – is to be extended. The underlying ideas for visualising XChange are postponed to Section 7.2.5.

Notes Along this thesis, the introduction of language constructs will be accompanied by the corresponding syntax rules. They explicitly state the valid combinations of XChange language constructs. Thus, the grammar for the language XChange is constructed in a stepwise manner. One of the most commonly used meta-syntactic notations for specifying the syntax of programming languages is the Backus-Naur Form (BNF) [112]. The BNF notation is a formal metasyntax used to express context-free grammars. There are many extensions to the BNF notation, one of them is the Extended Backus-Naur Form (EBNF) [143] notation. The EBNF notation used for defining the grammar of the XChange language follows the XML 1.1 Recommendation [39]. A short explanation of the EBNF notation follows. For a more detailed introduction into the EBNF notation, see [143, 39]. Terminals are symbols or words in the language, nonterminals are units representing a grammatically correct sequence of terminals. Productions are defined that specify the valid ways nonterminals can be replaced by terminals and other nonterminals. Extensions to BNF include: `|` denotes disjunction, `*` denotes that the preceding symbol or parenthesised expression may occur zero or more times, `+` denotes that the preceding symbol or parenthesised expression may occur one or more times, [ ] denotes optionality (note that instead square brackets, the symbol `?` is used so as not to confuse the optionality with the total specifications). (These extensions can be expressed in BNF by using extra productions.) *Note* that keywords are shown as quoted strings of characters (like `"and"`).

4.2 Events

The notion of *event* is defined as "something that happens at a given place and time"[1]; this explanation adapted from the WordNet 2.0[2] lexical database for the English language has (slight) variations that are used in different domains. In physics, an event is a change in the state of the world; in relativity theory, the fundamental observational entity is the event, i.e. a phenomenon located at a single point in space-time. In event-driven programming, "an event is a software message that indicates something has happened"[3]. For example, graphical user interface programs follow this programming paradigm, where small programs called event handlers are to be called in response to external events. Considering the public relations domain, an event is a tool for establishing and promoting a favourable relationship with the public; it can be organised as workshops, exhibitions, or panels that, in general, have a particular topic.

Section 1.3.1 already introduced the notion of *event* in an informal manner; it has also stressed the fact that one can conceive any kind of events in XChange. The above discussion on events does not offer a precise definition either, as might have been expected. It just states that a large number of events are conceivable and they correspond to many application domains. Thus, the (very general and abstract) definition of event fits perfectly into a reactive language for the Web. However, for realising reactivity on the Web, events require some representation for communicating their data between reactive programs on the Web and for processing their data by (local) event managers. In XChange, events are represented as XML documents. The language XChange has the ability to send, receive, and query events that are represented as *XChange event messages* (discussed in Section 4.3), i.e. messages containing any kind of event data represented as XML.

A *situation* is a combination of circumstances, that is a combination of events and other situations. Situations reflect particular states of the world, from low-level (such as ordered conjunctions of update operations on XML documents) to high-level ones (such as flight cancellations for which the airline does not grant an accommodation). Section 1.2 motivated the need for detecting situations that occur on the Web and recognised the ability to detect them as a requirement for Web reactive languages.

Not all events that have occurred on the Web, not all possible combinations of them (forming situations) are *of interest* for a Web site. Of interest are events and situations whose detection require an action to be automatically executed. At a moment in time, each Web site is interested in some classes of events and situations; these classes are determined by the rules of the Web site's reactive program (cf. Section 4.1.3). Modifying, deleting, or specifying new rules in a reactive program might entail other classes of events or situations to be of interest.

This section continues with presenting the two kinds of events supported by XChange, namely *XChange atomic events*, which reflect *events* (Section 4.2.1) and *XChange composite events*, which reflect *situations* (Section 4.2.2). Subsequently, Section 4.2.3 discusses the occurrence time for atomic events and composite events, respectively.

4.2.1 Atomic Events

An XChange *atomic event* is an event as it has been introduced in the previous section. XChange distinguishes between two kinds of atomic events: *explicit* events and *implicit* events. *Explicit events* are explicitly raised by a user or by a (predefined) XChange program at a Web site and sent to other Web sites through *event messages* (see Section 4.3). *Implicit events* are events that occur locally at a Web site (e.g. local updates of data or system clock events). Implicit events have also a representation as event messages, but (in general) a more simplified one than for explicit events. Implicit (or explicit) events raised and sent from a Web site to another become explicit.

The kinds of *atomic events* considered in XChange are presented in Table 4.1. An *update* executed or a *query* posed locally at a Web site are for XChange local events, i.e. raised at this Web site and processed at this Web site. *Transactional events* (transaction commit, transaction abort, transaction request) are local events needed as XChange supports the concept of transactions (cf. Section 4.1.5). *System events* (e.g. system clock events) are events that are coming from the encompassing "system" and might be useful to

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

Table 4.1: XChange Atomic Events

atomic events	explicit events (event messages)	
	implicit	updates
		queries
		transactional events
		system events

handle together with explicit and/or implicit events. A system event might be explicit or implicit, depending whether or not it is transmitted from one Web site to another.

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

4.2.2 Composite Events

XChange composite events reflect situations (introduced in Section 4.2). Composite events express temporal relationships between atomic events that have occurred on the Web. Also, they can express non-occurrence of some events while other events have occurred. XChange's (occurrences of) *composite events* are defined as answers to *composite event queries* (see Section 4.4.3). For understanding XChange's composite events and their representation, an elaborate discussion on XChange's event queries is needed. Section 4.4 elaborates on event queries in XChange. Thus, a more detailed discussion on XChange's composite events is postponed to Section 4.4.5.

4.2.3 Events' Occurrence Time

The occurrence time of (atomic or composite) events plays an essential role in determining whether to react or not to incoming events. For example, one might want to react to a certain class of atomic events but only if they are received before a given time point. For composite events, the occurrence time of atomic events is used in determining if a certain temporal order between them is met or not. Moreover, based on the occurrence time of events, atomic and semi-composed composite events can be released after a bounded time (see Section 4.4.3).

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

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

The work done in the active database field considers that each composite event has an occurrence time point, like atomic events do. The occurrence time of a composite event is the occurrence time point of the last received constituent atomic event. Queries against incoming events specifying e.g. that a particular event should occur twice during the occurrence of a composite event do not express the programmer's intuition. Thus, XChange follows another approach: XChange composite events do not have an occurrence time point, instead they stretch over time (they have a duration). Each XChange composite event has a *beginning time* and an *ending time*. In general, a composite event inherits from its components a beginning time (i.e. the reception time of the first received constituent event that is part of the composite event) and an ending time (i.e. the reception time of the last received constituent event that is part of the composite event). This is not the case for all composite events. Recall that composite events are defined as answers to composite event queries. Consider now a composite event query asking for non-occurrences of instances of event query *EQ* during a given time interval. Answers to such a composite event query do not have any constituent atomic events (as it is asked for non-occurrence); they have just a beginning and an ending time, time points determined by the given time interval. For each kind of XChange composite event queries, the beginning and ending time of their instances are explicitly specified in Section 4.4.3.

4.3 Event Messages

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

Event messages communicate event data between (same or different) Web sites. More concrete, XChange programs found at different Web sites raise events (cf. Section 1.3.5) and send their representations (i.e. event messages) to one or more XChange programs. For gaining the flexibility needed for implementing different kind of applications and for coping with event data having a complex and irregular structure, the XML format has been chosen for representing event messages. Section 2.2.1 has introduced the XML format and revealed the advantages of using it as a data interchange format.

Communication of event messages follows a push strategy (cf. Section 4.1.6), i.e. Web sites *inform* other Web sites about (implicit or explicit) events that have occurred on the Web. XChange excludes broadcasting of event messages on the Web (i.e. sending event messages to all sites of a portion of the Web), since indiscriminate sending of event messages to all Web sites introduces problems for a non-centrally managed structure such as the Web. Thus, in XChange each event message has a determined recipient Web site.

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

An XChange event message contains information about its sender Web site. This piece of information might be important for the recipient Web sites. Assume that Mrs. Smith is on vacation. Though, she would like her personalised organiser to be notified by her secretary if important problems occur in one of Mrs. Smith's projects. In such cases, not only the content of the event message but also the sender plays an important role in detecting the desired situation. However, applications not always need to make use of the sender address. Thus, the event language component of a reactive language should offer the ability to express partiality in queries to event data (i.e. to leave out parts of the event messages that are not of interest when specifying patterns to them). Section 4.4 shows that this language requirement is fulfilled by XChange.

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

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

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

4.3.1 Event Messages' Parameters

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

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

where

- `sender` is the URI of the Web site where the event has been raised, that is its representation as event message has been constructed and sent to one or more Web sites. The URI of the sender is determined and inserted into the event message by the Web site's event manager before sending it.
- `recipient` is the URI of the Web site that received the event message. As already explained, XChange excludes broadcasting of event data, implying that the recipient Web site(s) of an event message must be known before sending it. For an event message, at least one recipient needs to be given in the event message specification used to raise the event.
- `raising-time` is the time of the event manager of the Web site raising the event. This is the local raising time of the event message on the machine on which the event manager is running on.
- `reception-time` is the time at which the event manager of the recipient Web site receives the event message. This is the local reception time of the event message on the machine on which the event manager is running on. *Note* that the reception time of an event message might be before its raising time as no global time exists on the Web (proposals exist, such as [128], for models of an approximated global time base for distributed systems; however, this is not realistic in the largest distributed system – the Web).

  Event messages' time stamps (raising time and reception time) are given in XChange by using the ISO 8601 standard format for the representation of dates and times[4].
- `id` is an event message identifier given at the recipient Web site. Each event message gets at its reception such an identifier for uniquely identifying it in querying. The format and the method (e.g. counting received event messages each day and identifying them with the temporal specification of the actual day followed by the counter) used for event messages' identifiers are application specific. The current implementation of the XChange event manager uses positive integers for identifying event messages.
- `event data` is an XML element (having possibly complex structure) containing information about an event that has occurred. How the data about the event is represented is application specific. Reactive applications communicating through XChange event messages are not restricted to XML-based applications. For example, XML serialised RDF data can be also communicated between and processed by XChange programs.

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

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

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

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

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

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

Being XML documents, XChange event messages represent Xcerpt data terms (discussed already in Section 2.4.2) and thus, methods developed for querying persistent data can be also applied for querying incoming event messages. The importance of this note will be made clear in Section 4.4. The examples given in this section use the term syntax to represent event messages, but the reader should keep in mind that XChange programs communicate through XML documents that represent event messages.

Example 4.1 (XChange Event Message Notifying an Exhibition)
The following XChange event message is sent by `http://artactif.com` informing the travel organiser of Mrs. Smith about an exhibition of the painter G. Barthouil. *Note* the use of the `xchange` namespace for the keyword `event` and for the parameters of an XChange event message. *Note* also that the examples abstract away from a particular communication protocol. Here, `organiser` denotes the communication protocol used by a personalised organiser.

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

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

Example 4.2 (Nesting XChange Event Messages)
Mrs. Smith notifies a friend of her about G. Barthouil's exhibition. The following XChange event message is sent by Mrs. Smith's travel organiser and contains the received event message from the previous example:

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

Note that XChange messages are compatible with the messages and the "message exchange patterns" of SOAP (discussed in Section 2.3.3). XChange event messages can be seen as a very simplified form of SOAP messages, as only the minimum information is required (time stamps, sender, recipient, and id). However, XChange does not preclude the usage of more complex event messages' parameters. XChange applications can be implemented in such a way to construct such event messages and to understand their meaning properly.

4.3.2 Implicit Events' Representation

As explained in Section 4.2.1, implicit events are events that occur locally at an (XChange-aware) Web site. They become explicit if their representation is sent as event message to other Web sites. However, at the level of event representation no differentiated treatment should be applied for the two kinds of events. A uniform way of representing events is a premise for using the same event query language for querying them. Thus, XChange uses *event messages* for representing not only explicit events but also implicit events.

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

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

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

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

Example 4.3 (XChange Implicit Event Representation)
The following XChange event message gives a representation of an implicit event representing a modification of a Web resource; the event has occurred at Web site `http://xcerpt.org/xchange/`. The sender and recipient of the event message are not contained in the representation; the parameter `xchange:type` denotes the type of the implicit event, here an `update`.

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

The above given event message notifies about an insertion of a new scientific article on XChange; the term representing data about the article has been inserted as subterm of the `news` term.

4.4 Event Queries

For detecting situations that have occurred on the Web and require a reaction to be automatically executed, incoming event messages (i.e. representations of events that have occurred on the Web) need to be queried. Section 1.3.3 pointed out differences that exist between data of incoming events and data of Web resources, recognising that Web query languages are not suitable for querying event data. For this reasons, XChange offers *event queries* – queries against event data.

Real life situations, like the ones exemplified by the application scenarios of Section 1.2, need for their detection not just one event to occur, but (more often) more than one events to occur. Moreover, the temporal order of these (component) events and the specified temporal restrictions on their occurrence time points need also to be taken into account in detecting situations. Mirroring these practical requirements, XChange offers not only *atomic event queries* but also *composite event queries*. Thus, an XChange event query (symbol `EvQ`) is either an atomic event query (symbol `At_EvQ`) or a composite event query (symbol `Comp_EvQ`):

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

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

This section is structured as follows: Section 4.4.1 states explicitly essential traits of XChange event queries; their introduction intends to ease the understanding of most of the design decisions of the XChange event query language. Subsequently, Section 4.4.2 and Section 4.4.3 elaborate on atomic event queries and composite event queries, respectively. Language constructs are introduced for both kinds of event queries. The notion of answer to an event query is introduced in Section 4.4.5.

4.4.1 Essential Traits

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

Event Query vs. Web Query Event queries and Web queries serve different purposes – querying an incoming stream of events vs. querying Web resources; thus, they differ considerably in the communication strategy used, the querying capabilities, and the query processing. Event queries are fed with event data (to be queried) in a push manner, while querying Web resources is done using a pull strategy. Event queries query not only for single events but also for temporal combinations of incoming events, while Web queries lack constructs for dealing with temporal patterns over events. Event queries need to be evaluated in an incremental manner as events come in an stream-like manner and are not persistent. As Web resources' data are persistent, such requirements are not posed on the evaluation of Web queries. (Section 1.3.3 and Section 1.3.4 have offered a more elaborated discussion on differences between event queries and Web queries.)

Double Purpose XChange event queries have a double purpose: they are aimed for event detection and event data extraction. Event queries detect atomic and composite events (Section 4.2) that have occurred on the Web. XChange offers a considerable number of high-level constructs for expressing different kinds of event combinations (see Section 4.4.3). For extracting pieces of information from incoming events, *variables* are specified in event queries. Data items bound to the variables are to be subsequently used for raising events or executing updates.

Logical Variables Variables are place holders for the data, in the fashion of logic programming variables are. They require equality when occurring more than ones in an event query. *Note* that variables can be also bound to composite events' representations, not just to parts of atomic events. This ability is useful when e.g. a detected composite event needs to be sent to other interested Web sites (see Section 4.4.3).

Bounded Event Lifespan For processing XChange event queries (i.e. for detecting atomic or composite events as answers to them), events do not need to be kept forever in memory. Instead, event data are stored as long as they are needed for answering the event queries posed at a Web site. Moreover, the time for which data on any received event is kept in memory is bounded, i.e. the *event lifespan* is bounded. The notion of event history used in the literature [119, 141, 64] would be misleading in the context of XChange, as event data is not kept forever in memory and event queries do not query events received in the past.

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

Language Constructs The language XChange is rich in constructs for expressing different combinations of events to react upon. Their understanding requires some amount of time, though their usage eases considerably the programming task. During the design phase of XChange, a trade-of has been sought for between offering more language constructs and keeping their number as small as possible. Core language constructs (relative to the intended applications) can be used if they meet all application requirements.

4.4.2 Atomic Event Queries

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

Query Terms

The “simplest” XChange event query and, at the same time, the building block for more “complex” event queries (for detecting temporal combinations of events) is an Xcerpt query term. Its purpose, when posed against incoming events, is to detect single occurrences of events. Recall that Xcerpt query terms can be used for querying event data, as event data (i.e. event messages) represent data terms. Section 2.4.2 has introduced Xcerpt query terms and exemplified their core constructs on simple examples. A short recap of term specifications and query terms follows.

An *ordered term specification* (denoted by square brackets [ ]) expresses that the order of subterms is relevant, an *unordered term specification* (denoted by curly braces {}) expresses that the order of subterms is irrelevant and must not be kept. Ordered subterms are needed e.g. with texts like books. Unordered subterms are convenient with database-like data items. *Total* or *partial* (event and Web) query patterns can be specified. A query term q using a partial specification (denoted by *double* brackets [[ ]] or braces {{}}) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of q and that (2) might contain further subterms without corresponding subterms in q. In contrast, a query term q using a total specification (denoted by *single* brackets [ ] or braces {}) does not match with terms that contain additional subterms without corresponding subterms in q.

Example 4.4 (Simple XChange Atomic Event Query)
The following XChange atomic event query is a pattern that matches with all incoming events that a Web site receives (recall from Section 4.3 that XChange event messages have a root labelled `event` belonging to the `xchange` namespace).

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

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

- *descendant*, for expressing incompleteness in breadth (see Example 2.21);
- *without*, for expressing subterm negation (see Example 2.22);
- *except*, for leaving out certain subterms from a variable binding (see Example 2.23);
- *optional*, for specifying optional patterns inside query terms (see Example 2.24);
- other constructs whose explanation is found in [125].

Example 4.5 (Variables Inside XChange Atomic Event Queries)
The following XChange atomic event query detects event messages notifying a phone conference. The subject to be discussed and the time at which the phone conference should be held are of interest and thus are to be bound to the variables `S` and `T`, respectively.

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

For determining answers to atomic event queries and thus bindings for the variables, the event manager of an XChange-aware Web site attempts to match each incoming event received with the currently posed atomic event queries (which themselves may be part of composite event queries). Query terms are matched against event data (or Web resources' data) by means of a novel unification method called Simulation Unification [126, 125], which can handle querying constructs such as partial specifications, optional subterms, or negation of subterms. Informally, a query term q simulation unifies (or simply matches) a data term d if q's structure can be found in d. The outcome of simulation unifying q and d is a set of substitutions for the variables in q. XChange event queries ('event part') and Web queries ('condition part') are based on query terms and find substitutions for the variables that are then subsequently used in the 'action part' (event raising or transaction specification) of a rule.

Example 4.6 (XChange Event Message Notifying a Phone Conference)
Assume that the organiser of Mrs. Smith uses a rule containing the atomic event query of the previous example. An excerpt of an event message the organiser receives is given in the following using the term syntax:

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

The atomic event query of Example 4.5 detects the above phone conference notification; the evaluation of the atomic event query against the event message results in the following assignments for the variables: $S \mapsto$ `"Deliverable D5"` and $T \mapsto$ `"2005-04-25T14:00"`. Upon reception of other phone conference announcements having the specified pattern, the variables S and T will be bound to other data items.

Variables can be used not only *inside* event queries (e.g. variables `S, T` in Example 4.5), but also *outside* event queries. In the latter case, variables are to be bound to the whole event message that matches the atomic event query.

Example 4.7 (Variables Outside XChange Atomic Event Queries)
The following XChange atomic event query is a slight modification of the event query of Example 4.5. Both event queries detect phone conference announcements; the following one binds the variable `Msg` to the data term matching the given event pattern.

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

Upon reception of the event message of Example 4.6, the above atomic event query evaluates successfully and binds the variable `Msg` to the received event message (i.e. the substitution for variable `Msg` is exactly the data term of Example 4.6).

Posing Conditions on Atomic Event Queries As discussed already in Section 2.4.2, Xcerpt query terms may be further restricted by constraints (e.g. arithmetic expressions on variables occurring in the query term) in a so-called condition box, which has been introduced to source out all restrictions that are not pattern-based. Being Xcerpt query terms, atomic event queries inherit the condition box specification. The keyword `where` (as in Xcerpt) introduces such conditions on atomic event queries.

Example 4.8 (Conditions on XChange Atomic Event Queries)
The following XChange atomic event query detects event messages notifying a flight delay of more than forty five minutes.

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

Only events are detected that satisfy the given time constraint. More than one constraints on the variables occurring in the event query can be specified in the `where` clause. For a more detailed discussion on condition box specification, see Chapter 4.5.4 of [125].

Absolute Temporal Restrictions

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

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

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

Note that the production rules defining the nonterminal `Query_Term` are not given here; they are found in [125].

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

Time points in XChange are given using the ISO 8601 standard format for representing dates and times. "ISO 8601 advises numeric representation of dates and times on an internationally agreed basis."[5] Calendar date, week date, time of the day, and date and time can be represented using the standard. Representations begin with the largest element (e.g. year) followed by smaller elements (e.g. month followed by day). *Note* that years represent the Gregorian calendar's years. When a calendar date is followed by the time of the day in the ISO 8601 representation, the capital letter T is used to separate the date and time components. For example, 2005-07-07T14:05:00 represents five minutes after two o'clock in the afternoon of July 7, 2005.

By using the ISO 8601 format for specifying dates and times, it is assumed that all parties use the same calendar – the Gregorian one. In order to facilitate communication between personalised reactive systems whose owners use different calendars, means for defining calendars and reasoning with calendar data are needed. Richer temporal specifications are conceivable in XChange. This can be achieved e.g. by integrating into XChange a calendar and temporal type system such as CaTTS (Calendar and Temporal Type System) [51].

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

Example 4.9 (XChange Atomic Event Query Specifying Temporal Restriction)
The following XChange atomic event query detects only such events whose representation has been received before 2005-07-07T14:00:00 and, of course, matches the given pattern.

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

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

Example 4.10 (XChange Atomic Event Query Detecting Discounts)
An XChange atomic event query that detects insertion of discounts for flights from Munich to Paris that are received as notifications before July 7, 2005 is given next.

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

Note that insertions can be notified by using other structure for event messages. However, the update specification that has been used to perform the insertion can not be sent as content of event messages as update specifications are not data terms. A detailed introduction into XChange update specifications is given in Section 4.6.

4.4.3 Composite Event Queries

The capability to detect and react to *composite events*, e.g. sequences of events that have occurred possibly at different Web sites within a specified time interval, is needed for many Web-based reactive applications. However, (to the best of our knowledge) existing languages for reactivity on the Web do *not* consider the issues of detecting and reacting to such composite events ([29] considers detecting composite events, but XChange's notion of composite events goes beyond their notion, cf. Section 3.2.1). One of the novelties introduced by XChange is the processing of *composite events*. To this aim, XChange offers *composite event queries*.

Composite event queries are specified by means of atomic event queries combined using XChange composite event query constructs. XChange offers a considerable number of such constructs along two dimensions: *temporal restrictions* and *event compositions*. This section gives an introduction into XChange constructs for composite event queries; their syntax and informal semantics are given here, the formal definition of the semantics is postponed to Section 5.1.1.

Temporal Restrictions

The role of temporal restrictions on composite event queries is twofold: they specify interest in events that occur in a given time interval or have a given duration, and ensure that event data can be released after a bounded time (this is realised by using *legal composite event queries*, notion introduced in Section 4.4.4).

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

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

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

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

- `CEQ in [Time_Point`$_1$ `.. Time_Point`$_2$`]` if the beginning time of `c` is greater than or equal to `Time_Point`$_1$ and the ending time of `c` is less than or equal to `Time_Point`$_2$;
- `CEQ before Time_Point` if the ending time of `c` is less than or equal to `Time_Point`.

  Clearly the above stated temporal conditions on `c` are not enough for detecting `c` as an answer to the composite event query `CEQ`.

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

```
Comp_EvQ ::= Comp_EvQ  "within" Duration

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

XChange requires every composite event query to be accompanied by a temporal restriction specification. This makes it possible to release each (atomic or semi-composed composite) event (i.e. to release event queries' answers or partial instantiations of them) at each Web site after a finite time. A detailed discussion on the temporal restrictions that should accompany event query specifications for releasing event data is postponed to Section 4.4.4. Thus, language design enforces the requirement of a bounded event lifespan and the clear distinction persistent vs. volatile data.

Event Composition

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

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

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

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

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

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

Also, $beginning\_time(comp\_event)$ and $ending\_time(comp\_event)$ are used for denoting the beginning and ending time, respectively, of composite event $comp\_event$.

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

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

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

Example 4.11 (XChange Event Query Specifying Conjunction (1))
The following event query specifies interest in the occurrence of pairs of events whose contents match the atomic event queries $a\{\{\}\}$ and $b\{\{\}\}$, respectively:

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

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

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

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

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

Example 4.12 (XChange Event Query Specifying Conjunction (2))
The following event query is a slight modification of the Example 4.11 (above), where the variable X is to be bound to the content of the incoming event messages that match the two atomic event queries.

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

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

Example 4.13 (XChange Event Query Specifying Conjunction)
Mrs. Smith wants to visit an exhibition of G. Barthouil on a rainy day. The next XChange event query is used to detect the conjunction of the exhibition notification and the desired weather forecast notification that are sent by appropriate Web services.

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

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

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

A total specification (i.e. single square brackets) expresses that the answer to such a composite event query contains only the instances of the component event queries. Between the instances of the specified event queries other events might occur; they neither influence the successful evaluation of the event query, nor are part of the answer.

A partial specification (i.e. double square brackets) for a temporally ordered conjunction event query expresses that the answer contains besides the events that answer the component event queries also all events that have occurred in-between. The practical need for total *and* partial specifications for such event queries has been already motivated by the examples of Section 4.1.7.

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

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

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

Example 4.14 (XChange Event Query Specifying Temporally Ordered Conjunction (1))
The following event query specifies interest in the occurrence of sequences of events having content with label a and b, respectively.

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

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

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

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

Note the difference to the answers of the event query of Example 4.11: sequences $b\{c\}$, $a\{d\}$ and $b\{c\}$, $a\{e\}$ are not answers to the event query as the temporal order between these events is not the desired one.

Example 4.15 (XChange Event Query Specifying Temporally Ordered Conjunction (2))
The following event query specifies interest not only in sequences of events having content with label a and b, respectively, but also in all events that have occurred in-between.

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

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

Example 4.16 (XChange Event Query Specifying Temporally Ordered Conjunction (3))
An example of an `andthen` event query that collects only events with a particular pattern:

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

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

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

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

Example 4.17 (XChange Event Query Specifying Temporally Ordered Conjunction)
The next XChange event query is used to detect the notification of a flight cancellation and afterwards, within two hours from its reception, the detection of a notification informing that the accomodation is not granted by the airline.

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

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

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

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

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

Example 4.18 (XChange Event Query Specifying Inclusive Disjunction (1))
The following event query specifies interest in the occurrence of events having content with label a or b.

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

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

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

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

Example 4.19 (XChange Event Query Specifying Inclusive Disjunction)
After Orange, Mrs. Smith wants to visit Arles and Nîmes. The next city to visit is chosen depending on the notification of train tickets and hotel reservation made by appropriate services.

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

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

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

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

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

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

Example 4.20 (XChange Event Query Specifying Exclusion (1))
The following event query specifies interest in the non-occurrence of *c*-labelled events during occurrence of sequences of events labelled *a* and *b*, respectively.

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

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

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

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

Example 4.21 (XChange Event Query Specifying Exclusion (2))
The following exclusion event query is a slight modification of the previous example where some of the component atomic event queries are augmented with variables.

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

Consider that the above given event query is to be evaluated against the excerpt of the incoming event stream given in the previous example. Upon reception of $a\{e\}$ the `andthen` event query is partially evaluated and the variable gets a binding, the substitution $\sigma_1 = \{X \mapsto e\}$ is obtained. The reception of $e\{f\}$ does not influence the evaluation of the event query. Upon reception of $a\{d\}$ another instance of the `andthen` event query is partially evaluated and a possible binding for the variable is found, the substitution $\sigma_2 = \{X \mapsto d\}$ is obtained. The event $c\{e, f\{g\}\}$ matches the event query whose exclusion is of interest; the result of $c\{\{var\ X\}\} \preceq c\{e, f\{g\}\}$ gives two possible bindings for the variable, $\Sigma_3 = \{\{X \mapsto e\}, \{X \mapsto f\{g\}\}\}$. Receiving $b\{f\}$ determines two answers to be found for the `andthen` event query. Now, the exclusion event query can be evaluated. Recall that variables used in event queries require equality when occurring more than ones in a query. Thus, the whole event query is evaluated successfully only once against the given incoming event stream, with the sequence $a\{d\}$, $b\{f\}$ (no c-labelled events having a child d are received, thus the event query is successful). The variable substitution obtained is $\sigma = \{X \mapsto d\}$ ($\sigma_2 \wedge \neg\Sigma_3$).

Example 4.22 (XChange Event Query Specifying Exclusion (3))
The following exclusion event query detects non-occurrence of c-labelled events within the given time interval.

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

Assume that no c-labelled events have occurred within the specified finite time interval; at time point `2005-05-22T20:00:00` the exclusion event query evaluates successfully. As no c-labelled events are received, no binding for the variable X is found. Thus, the variable X can not be further used in the reactive rules having as event part the above given exclusion event query.

Variables occurring in the event queries whose exclusion is of interest (i.e. event queries specified after the keyword `without`) need to have at least one *defining* occurrence in the (whole) event query in order to be further used in an event query or other parts of XChange rules. Each variable occurrence in XChange rules is associated with a *polarity* for determining whether a variable occurring in the event part (or condition part) of the rule can be used in the condition and action part (or, just in the action part, respectively) of the rule or not (i.e. determining rules' *range restriction*). A *negative* polarity of a variable occurrence expresses a defining occurrence of the variable. A *positive* polarity expresses a non-defining variable occurrence. The polarity of Xcerpt query terms (defined in [125]) is extended for XChange event queries. The polarity of event queries and the range restriction of XChange rules are postponed to Section 4.8.4. In Example 4.21 the first occurrence of the variable X has positive polarity as it occurs inside the event query whose exclusion is of interest, the second occurrence of X has a negative polarity. Thus, the variable can be used outside the event query (e.g. in complex event queries having as one of the components the exclusion event query). In Example 4.22 the variable X occurs ones with positive polarity, meaning that the variable can not be used outside the given exclusion event query.

Example 4.23 (XChange Event Query Specifying Event Exclusion)
The following XChange event query detects if the notifications of two online reservations made on 10th of July 2005 are not received within ten days.

```
without {
  and {
      xchange:event {{
              flight-reservation-notification {{ }}
       }},
      xchange:event {{
              hotel-reservation-notification {{ }}
       }}
  }
 } during [2005-07-10..2005-07-20]
```

Occurrences *Occurrences* constructs for event queries refer to the number of times an event query instance should occur or should be repeated to be of interest, or to the position that events of interest should have in the incoming event stream. The occurrences constructs supported by XChange (and explained in the following) are (1) *quantifications*, (2) *repetitions*, and (3) *ranks*.

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

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

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

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

Example 4.24 (XChange Event Query Specifying Quantification (1))
The following event query specifies interest in the reception of at least three messages from the secretary within the specified time interval. Also, the subject of the messages are of interest (e.g. for using them in the action part of the rule having the following event part).

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

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

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

The existential quantified variables' specification follows the occurrence specification (`times M`). The keyword `any` precedes the list of the existential variables used in an event query. Variables not declared as existential quantified do require equality when occurring more than ones in an event query. Declaring a variable as existential quantified in an event query applies to all its occurrences in the component event queries (the property of being existential quantified for a variable is inherited in a top down manner to component event queries).

Example 4.25 (XChange Event Query Specifying Quantification (2))
The following event query specifies interest in the reception of at least three messages from the secretary within the specified time interval.

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

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

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

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

2. Repetitions are used for detecting e.g. every second, forth, sixth, and so on, instances of a specified event query in a given time interval or between occurrences of other event query instances.

The keyword `every` introduces such event queries in XChange. The following grammar rules define repetition event queries:

```
Comp_EvQ ::= "every" Nr ("any" Vars)?  "{" EvQ "}"
```

Example 4.26 (XChange Event Query Specifying Repetition (1))
The following event query detects every second instance of the specified atomic event query $a\{\{var\,X\}\}$.

```
every 2 {
    a {{ var X }}
  }
```

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

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

The event $a\{b,c\}$ matches the atomic event query $a\{\{var\,X\}\}$, thus a first instance of it is found; the result of simulating the atomic event query into the event $(a\{\{var\,X\}\} \preceq a\{b,c\})$ is the substitution set

$\Sigma_1 = \{\{X \mapsto b\}, \{X \mapsto c\}\}$. The events $b\{e\}$, $g\{h\}$ do not influence the answer to the event query. Upon reception of $a\{c,d\}$ the repetition event query has a first answer (i.e. reactive rules having as event part the given event query fire); the event $a\{c,d\}$ matches the atomic event query and the result of $a\{\{var\ X\}\} \preceq a\{c,d\}$ is the substitution set $\Sigma_2 = \{\{X \mapsto c\}, \{X \mapsto d\}\}$. Thus, the whole event query evaluates successfully with substitution $\sigma_{answer_1} = \sigma_1 \cap \sigma_2 = \{X \mapsto c\}$.

The event $a\{f\}$ matches the atomic event query and a new substitution for the variable is found, $\sigma_3 = \{X \mapsto f\}$. If an a-labelled event with a child f is received at some point in the future, then the repetition event query would evaluate successfully with substitution σ_3.

Upon reception of $a\{d,e\}$ the repetition event query is evaluated successfully for the second time against the given event stream (the two answers of the event query are written in red, the events matching the atomic event query but being no answers to the whole event query are written in blue). $a\{\{var\ X\}\} \preceq a\{d,e\}$) resulted in the substitution set $\Sigma_4 = \{\{X \mapsto d\}, \{X \mapsto e\}\}$. The event represents the second instance of the repetition event query having as substitution $\sigma_{answer_2} = \Sigma_2 \cap \Sigma_4 = \{X \mapsto d\}$ (the first corresponding instance was $a\{c,d\}$).

Example 4.27 (XChange Event Query Specifying Repetition)
Mrs. Smith wants to quit slowly smoking so she answers only to every second call from her colleague suggesting a smoking break. Such an event query can be specified in XChange and is given next.

```
every 2 {
    xchange:event {{
       xchange:sender {organiser://institute/myColleague/},
       break-for-a-smoke {{
         info {"Join me for a cigarette!"}  }}
    }}
  } in [2005-03-01..2005-04-01]
```

3. Ranks are used to detect instances of a specified event query having a given rank (or position) in the incoming stream of events. For determining the position of an event query instance, the instances that are found for the event query are counted either until the desired position is reached or until the given monitoring time interval ends (for determining the last instance of the event query). For this, the reception time point of atomic event query instances and/or the beginning and ending time points of composite event query instances are used.

The position of each of the instances of an atomic event query in an incoming event stream is rather easy to determine. The first answer to the atomic event query (i.e. the first event received after event query registration that matches the atomic event query) gets position 1 in the event stream. Assume that at a moment in time t the position of the last received instance of the atomic event query is p. The next received event e (reception time of e is greater than t) matching the atomic event query gets position $p+1$. Considering two events e_p and e_{p+i} having positions p and $p+i$ ($1 \leq p$, $1 \leq i$), respectively, it holds that $reception\_time(e_p) < reception\_time(e_{p+i})$.

Different approaches are conceivable for determining the position of composite event query instances in an incoming event stream. This is illustrated by means of a simple example.

Example 4.28 (Position of Composite Events in an Incoming Event Stream)
Recall the event query of Example 4.11:

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

Assume that after registering the event query at a Web site the following events are received (events are indexed by positive numbers representing an abstraction of the reception time points, a short notation intended to simplify the example)

$a_1\{1\}, b_2\{1\}, a_3\{2\}, a_4\{3\}, b_5\{2\} -- >$

Posing the above given event query against the excerpt of the incoming event stream entails six answers to be obtained. The first instance of the event query has as components the events $a_1\{1\}$ and $b_2\{1\}$; it gets position 1. The second instance has as components $b_2\{1\}$ and $a_3\{2\}$; it gets position 2. The third instance is composed of $b_2\{1\}$ and $a_4\{3\}$; it gets position 3.

Upon reception of $b_5\{2\}$ the event query has other three answers, as the event completes the sequences formed of $a_1\{1\}$, $a_3\{2\}$, and $a_4\{3\}$, respectively. Because the ending time of these three answers is the (same) time point $t = 5$, the beginning time points of the answers need to be used for determining the position of the instances in the event stream. Thus, the answer composed of $a_1\{1\}$ and $b_5\{2\}$ gets position 4, the one composed of $a_3\{2\}$ and $b_5\{2\}$ gets position 5, the answer composed of $a_4\{3\}$ and $b_5\{2\}$ gets position 6. But why not use just the ending time of the composite event query instances in determining their position in the event stream? This would mean that the last three answers to the event query get position 4. Both approaches are conceivable. The most suitable one should be chosen depending on the applications to be developed.

The example given above shows that composite event query instances (i.e. composite events) may be ordered in different ways based on their beginning and/or reception time points. Thus, for keeping the language as clear and simple as possible, XChange is not committed to a single approach to ordering composite events on the time axis of incoming events. Instead, XChange event queries specifying ranks are restricted only to atomic event queries. Though, the language covers a large range of application domains and scenarios and does not preclude its extension to detecting composite events having a given rank.

The keywords `withrank` and `last` introduce such event queries in XChange. The last instance of an event query can be determined either at the end of a time interval or at a successful evaluation of a composite event query. Thus, for offering a uniform syntax and semantics of language constructs, the keyword `during` introduces such a time interval or composite event query. The following grammar rules define ranks event queries in XChange:

```
Comp_EvQ ::= "withrank" Nr ("any" Vars)?  "{" At_EvQ "}"
          |  "last"   "{" At_EvQ "}"  "during"  "{" Comp_EvQ "}"
          |  "last"   "{" At_EvQ "}"  "during"  Finite_Time_Interval
```

Example 4.29 (XChange Event Query Specifying Ranks (1))
The following event query detects non-empty, f-labelled events that represent the last ones received during instances of a temporally ordered conjunction event query.

```
last {
  f {{ var X }}
 } during {
          andthen [
            a {{ }},
            b {{ }}
            c {{ }}
          ]  }
```

Assume that after registering the event query at a Web site the following events are received:

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

Posed against the above given excerpt of an event stream, the event query evaluates successfully; the answer to the event query is composed of $a\{g,h\}$, $b\{h\}$, $f\{d,e\}$, and $c\{d\}$ (the atomic event queries and their instances that are part of the answer to the whole event query are written using the same colours). The substitution set for the variable used is $\Sigma = \{\{X \mapsto d\}, \{X \mapsto e\}\}$. *Note* that the last instance of the desired event query is not $f\{\}$, as f-labelled events are sought for that have at least one child element ($f\{var\,X\} \preceq f\{\}$ fails).

Example 4.30 (XChange Event Query Specifying Ranks)
As the airline might send several delay notifications for a flight, the following event query can be used for detecting the last notification that occurred during the first signalling of delays and the notifications announcing the boarding time.

```
last {
   xchange:event {{
     delay-notification {{
        var F,
        expected-departure-time { var DT }
     }}
   }}
 } during {
          andthen [
            withrank 1 {
               xchange:event {{
                 xchange:sender { "http://airline.com" },
                 delay {{ var F -> flight-number {{ }} }}
               }}
            },
            xchange:event {{
              boarding-time {{ var F, begin { var BT }  }}
            }}
          ]  }
```

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

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

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

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

Example 4.31 (XChange Event Query Specifying Multiple Inclusions and Exclusions (1))
The following event query specifies an exclusive disjunction of *a*-labelled and *b*-labelled events. The occurrence of such events is of interest during instances of a conjunction event query.

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

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

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

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

Multiple inclusions and exclusions event queries specify interest in occurrence of event query instances and non-occurrence (event exclusion) of other event query instances; one does not know beforehand which event queries will be answered and which not. Thus, it might be the case that not all variables used in the event queries will have bindings. Recall the discussion on variable substitutions for the case of exclusion event queries. Similar, variables occurring in the event queries whose inclusion or exclusion is of interest (i.e. event queries specified after the keyword `of`) need to have at least one *defining* occurrence in the (whole) event query in order to be further used in an event query or other parts of XChange rules. In Example 4.31 the variable Y has a defining occurrence as it will be bound when evaluating the conjunction event query (which determines the monitoring time interval). The occurrence of variable X is non-defining and thus it can not be further used in the rule having as event part the example event query. (Though, if the event query is part of a more complex event query, the variable X might have a defining occurrence in other parts of the complex event query.)

One might argue that the language is too restrictive because of the requirement of at least one defining occurrence for the variables that need to be used in other parts of an XChange rule. Another approach would consist in introducing a kind of optionality specification for the variables that do not have a defining occurrence in the event query; the specification accompanies these variables in the condition and action part of XChange rules. A default value for the variable marked "optional" needs also be specified; this value is to be used when no binding for the variables resulted from a successful evaluation of the event query.

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

```
var CE ⟶ Comp_EvQ
```

The variable CE is to be bound to the answers found for *Comp_EvQ*. Answers to composite event queries contain all atomic events that are used for answering the event query, they are sequences of atomic events whose representation is an XML document. (A more detailed discussion on answers to XChange event queries and their representations is given in Section 4.4.5.) Thus, variables occurring in (more precisely, inside or outside) XChange event queries are bound to data terms.

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

Example 4.32 (Nesting XChange Event Queries)
The following example gives a composite event query for detecting occurrences of a flight cancellation, where the airline does *not* grant an accommodation. For this purpose, a temporal ordered conjunction construct, the exclusion construct, and temporal restrictions are combined.

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

Posing Conditions on Composite Event Queries As for atomic event queries, variables occurring in composite event queries can also be constrained with conditions specified in a `where` clause. (Recall that only non-structural conditions are to be specified, structural conditions are given through event query patterns.) The following grammar rule defines composite event queries with condition box specification (for a detailed explanation of `Condition`, see Chapter 4.5.4 of [125]):

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

Example 4.33 (Conditions on XChange Composite Event Queries)
Consider the simple composite event query:

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

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

On Introducing Other Constructs for Composite Event Queries (Discussion) Composite event queries specify temporal relationships between classes of events; composite event query instances (composite events) have a beginning and an ending time (time points that can be seen as the starting and the ending point of a time interval). Thus, determining the temporal order of two (or more) composite event query instances can be reduced to determining the relationship between the time intervals formed from the beginning and ending time of the these instances. The possible relationships between time intervals have been described and represented in a hierarchical manner by James F. Allen [13]. The thirteen possible relationships between time intervals have provided a basis for the development of the XChange constructs for detecting composite events. Let's look at these temporal relationships between time intervals to determine which of them *are* and which *are not* covered by the constructs offered by XChange.

Let E_1 and E_2 be two finite time intervals. For expressing the temporal relationships the notation introduced in [14] is used.

- The relationship $E_1 : before\ E_2$ is mirrored by the temporally ordered conjunction event queries (introduced by the keyword `andthen`).

- The relationships E_1 : *starts* E_2 and E_1 : *finishes* E_2 can be expressed in XChange by using the temporally ordered conjunction event queries with constraints on the beginning and ending times of the instances, or by requiring equality between first component atomic events. *Note* that for two composite event query instances to have the same beginning time (meaning of : *starts*), either the first component atomic event is the same for both instances (in the setting of XChange one single event is received at a point in time) or the beginning time of the first component atomic event of an instance is the start time of the finite time interval in the `during` specification of the other instance (for event exclusion). Similar, : *finishes* of XChange composite events can be explained.

- An approach similar to that discussed previously can be applied for E_1 : *equals* E_2.

- The relationship E_1 : *during* E_2 is mirrored by the `times` construct for event queries where one occurrence ($Nr = 1$) of a composite event (corresponding to E_1) is detected *during* occurrences of other composite events (corresponding to E_2):

```
Comp_EvQ ::= "times" 1 "{" EvQ "}"  "during"   "{" Comp_EvQ "}"
```

- The counterpart of the temporal relationship E_1 : *overlaps* E_2 is represented by querying for overlapping of composite event query instances on the time axis of the incoming events. It can be expressed in XChange but not that simple as e.g. expressing : *before*, as no construct for overlapping of composite event query instances exists in XChange (clearly, the binary case is more simpler than detecting the overlapping of n composite events, $n > 2$). Use cases developed for XChange do not bear evidence for the exigency of such a construct. However, for applications where such a construct would ease considerably the programming task, the event query language of XChange could be extended with:

```
Comp_EvQ ::= "overlap"  "["  Comp_EvQ  ("," Comp_EvQ)+  "]"
          |  "overlap"  "{"  Comp_EvQ  ("," Comp_EvQ)+  "}"
```

  Total specifications denote that besides overlapping of composite event query instances, the order of their occurrence need to conform to the event queries' specification.

- The temporal relationship E_1 : *meets* E_2 does not have a corresponding construct for event queries in XChange; the discussion on : *overlap* also applies to : *meets* for event queries. However, a construct like `meets` would be more interesting when considering different time granularity for the "meeting point" on the time axis. For example, one could be interested in sequences of composite events $Comp\_EvQ_1$ and $Comp\_EvQ_2$ that *meet* with time granularity *week*; the informal semantics of such an event query is that $ending\_time(comp\_ev_2) - ending\_time(comp\_ev_1) \leq week$, where $comp\_ev_i$ is instance of $Comp\_EvQ_i$, $i = 1, 2$ and *week* is defined as a duration of 7 days (defined by means of a temporal system integrated into XChange). Different approaches to the semantics of such event queries are conceivable. For extending XChange event queries with a construct mirroring : *meets*, the grammar rules could look like:

```
Comp_EvQ ::= "meets"  "["  Comp_EvQ  ("," Comp_EvQ)+  "]"  Time_Spec
          |  "meets"  "{"  Comp_EvQ  ("," Comp_EvQ)+  "}"  Time_Spec
```

  where *Time_Spec* is the time granularity specification and depends on the temporal system used.

One of the ideas during the development of XChange that never got materialised was to introduce a branching construct – an *if-then-else* for event queries. The informal semantics of such a construct is: *if* querying for a class of events A detects an instance of it, *then* query for events B, *else* query for events C. Such an event query can be expressed by means of two rules in XChange (one querying for conjunctions of A and B, and one for conjunctions of *without* A and C). For application scenarios where the number of alternatives to be queried is small and the event queries shared by more than one rules are not complex, the existing language constructs would do with no considerable programming effort. However, applications can be found where an *if-then-else* construct for event queries might be very useful. If considered necessary

XChange could be extended for offering such a construct for event queries. A generalisation of this control construct (i.e. a *case* construct) can also be conceivable for further extensions of the language. Here are the grammar rules defining the extensions to the language syntax:

```
Comp_EvQ ::= "if"  "{" EvQ "}"   "then" "{" EvQ "}"   ("else" "{" EvQ "}")?
          |  "case"  "{" EvQ "}"  "then" "{" EvQ "}"
                    (","  "{" EvQ "}"  "then"  "{" EvQ "}")*
                    (","  "else" "{" EvQ "}")?
```

It is not that clear which kind of constructs should necessarily be included into a reactive language developed not only for a single kind of applications, but trying to cover different classes of applications. Developing use cases for a language entails introduction of new language constructs and (perhaps) removing others; it also reveals the limits of a language. Moreover, a tradeoff between the expressive power of the language and the ease of its usage needs to be found in designing a language. The design of XChange event queries (and in fact of the whole language XChange) has aimed at introducing powerful constructs that ease the programming task; though, efforts have been put in to avoid XChange to become a *giant-size language* that can everything but nobody is going to use it.

4.4.4 Legal Event Queries

Recall the statement *"XChange event queries are such that volatile data remains volatile"* given in Section 4.1.8. An essential trait of event queries (cf. Section 4.4.1) is that they ensure that data of no event is kept forever in memory; that is, the event lifespan is bounded. Though, (composite) event queries as introduced in the previous section can need an unbounded lifespan for events.

Example 4.34 ("Illegal" XChange Composite Event Query)
The following composite event query specifies interest in a-labelled events followed by b-labelled events.

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

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

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

XChange assumes no consumption of events, thus the same event may be part of more than one instance of a composite event query. After detecting the instance of the above `andthen` event query composed of $a\{e\}$ and $b\{f\}$, the atomic event $a\{e\}$ needs to be kept in memory for waiting to other b events to occur. Upon reception of $b\{h\}$, using the events kept in memory, another instance of the event query is detected (the instance composed of $a\{e\}$ and $b\{f\}$). The next event received matches the atomic event query $a\{\{\}\}$, thus needs to be kept in memory for b-labelled events that will possibly be received in the future. (If instead of an `andthen` event query an `and` event query is used, besides a-labelled events also b-labelled events need to be kept in memory.) As the event manager can not predict which kind of events will be received, event data needs to be kept forever in memory. For avoiding this, restrictions on composite event queries are posed.

XChange (composite) event queries are restricted to so-called *legal event queries* that can be evaluated with bounded event lifespan. A formal proof of this statement is given in Section 5.1.1. Legal event queries have the promised trait of keeping the clear cut between persistent and volatile data.

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

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

```
Legal_EvQ  ::= At_EvQ
             | LC_EvQ

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

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

Example 4.35 (XChange Composite Event Query)
The following composite event query is not legal with respect to the definition above. ($T2$ represents a time point.)

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

Though, it can be rewritten as a legal (cf. supra) composite event query:

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

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

```
LC_EvQ  ::= Inf_EvQ

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

On Restricting Composite Event Queries (Discussion) Other approaches for ensuring a bounded event lifespan are also conceivable. One can think of using a *system timeout for events* meaning that all events older than a duration *d* are to be deleted. However, system timeouts do not always express users' requirements, as event lifespans are application dependent. However, the restrictions posed on event queries in XChange may be lifted for applications where other kind of restrictions are more suitable. For example, *predicting classes of events* that will be received based on those received so far is an interesting research issue whose investigation would probably lead to another solution for keeping the event lifespan bounded.

4.4.5 Answers to Event Queries

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

Considering just the atomic events answering event queries is not enough, the substitution sets of answers to event queries play an important role – they represent communication means between the components of XChange reactive rules: The substitutions for the variables occurring in an event query restrict the possible substitutions for the variables occurring in the Web query and action specification of the same XChange reactive rule. The substitutions provide data for performing the desired actions, for constructing notifications to be sent to other Web sites and for constructing new data to be inserted into (local or remote) Web resources' data. The *maximal* substitution set for *all* variables having at least one defining occurrence in the event query is considered for the answer to an XChange event query. Substitutions for *all* variables are of interest so as to be able to group (e.g. by using the grouping construct `all`) the substitutions when used in the action part of XChange rules. Intuitively, a substitution set Σ answering the event query *EvQ* is *maximal*, if there exists no substitution set Υ answering *EvQ* such that Σ is a proper subset of Υ. A more general and formal definition of maximal substitution sets can be found in [125] (Section 7.3, Definition 7.1 on page 147).

Answers to Atomic Event Queries An answer to an atomic event query is made of

(i) the *atomic event* whose representation (as event message) matched the event query (and occurred in the given time interval, if a temporal restriction has been specified). Atomic events, are represented as XML documents – they are XChange event messages. A DTD for the representation is given in Section 4.3.

(ii) the (maximal) *substitution set* for all variables with at least one defining occurrence in the event query; the substitutions are the result of matching (*simulation unifying*) the atomic event query with the atomic event the answer contains.

Answers to Composite Event Queries An answer to a composite event query is made of

(i) a *sequence of atomic events* that have occurred and have been used to answer the composite event query; it contains *all* atomic events that participated to answering the event query.

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

An *XChange composite event* is a sequence of atomic events that altogether are used for answering a composite event query. Thus, an answer to a composite event query is made of a composite event and a substitution set (for the variables occurring in the composite event query the composite event answered). Composite events should also be representable as XML documents, just like atomic events; this allows further processing of composite events at local and remote Web sites. A composite event is represented as a (flat) sequence (with an artificial root to make valid XML) of all atomic events that were used for answering the composite event query. The atomic events are ordered by their reception times. The first and last child elements of a composite event's representation are its beginning time and ending time, respectively. A DTD for the representation of XChange composite events is given next. `xchange:event` represents an XChange event message; a DTD for it is given in Section 4.3.

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

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

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

Variables can be bound to composite events, or more exactly to their representation as XML documents. This can be achieved by restricting a variable to the answers to a composite event query – by writing *var* $A \rightarrow Comp\_EvQ$. The bindings for variable A are the composite events answering the composite event query $Comp\_EvQ$; an example of such a binding is given in the next example, which shows an answer to a quantification event query.

Example 4.36 (XML Representation of a Composite Event)
The example shows an answer to the composite event query specifying quantifications; the event query has been given as Example 4.25.

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

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

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

Example 4.37 (XML Representation of a Composite Event (Possible Approach))
Consider the following composite event query ($T1$ denotes a specification of a time point):

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

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

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

An advantage of having answers mirroring the structure of composite event queries is that Web sites receiving composite events data can determine very easy the whole or the components of the composite event queries they answered. A disadvantage of this approach is that same atomic events might be found several times in the representation of a single composite event. However, a flat sequence for representing composite events is better. It is simpler and more intuitive for users, since no knowledge of the query structure is required. It leads to an easier definition of declarative semantics (see Section 5.1.1), due to the similarity between sets and sequences. Finally, it is desirable for a query language to have similar input and output — and the input of event queries is a collection of atomic events arriving sequentially. In principle, this allows using the answer to a composite event query as the input to another event query.

4.5 Web Queries

Web queries are queries to persistent data (i.e. data of Web resources); they represent the 'condition part' of XChange reactive rules (cf. Section 1.3.7 and Section 4.8). Web queries determine if certain conditions hold (e.g. a person making a rental order is one of the clients of the rental firm, or the notification of a flight cancellation regards Mrs. Smith flight) and gather data as variable bindings that is needed for performing the desired actions (e.g. insertion of a new client in the database with the informations received through the rental order, or booking an overnight stay for Mrs. Smith where the flight date is used).

In XChange, Web queries are expressed using the Web and Semantic Web query language Xcerpt [125] that has been introduced in Section 2.4.2. An XChange Web query is an Xcerpt query, that specifies conjunctions or disjunctions of query terms. Query terms are used here for specifying patterns for the data to be queried augmented with variables for selecting data of interest. Variables bindings can be restricted to given patterns. Non-structural conditions on variables are specified in a `where` clause attached to the query terms or the whole query.

Web queries (Xcerpt queries) can query persistent data directly or by querying views constructed by means of deductive rules (Xcerpt construct-query rules). A resource specification inside an Xcerpt query gives the Web resources to be queried. If no resource specification is given, the Xcerpt query is posed against the data constructed by means of the Xcerpt construct-query rules contained in the same XChange program; thus, complex querying problems can be elegantly solved by using views over multiple, heterogeneous data sources. Section 4.8.3 gives an example of an Xcerpt rule constructing such a view.

This section does not elaborate more on Web queries, as they have been developed as part of the project Xcerpt and the design principles and core constructs of the language Xcerpt have been introduced in Section 2.4.2. Section 4.8 shows how Web queries are used together with event queries and action specifications in XChange so as to give reactive rule specifications. For more information on the query language Xcerpt used for specifying Web queries, see [125, 55, 53, 54, 126].

4.6 Update Patterns

As explained in Section 3.2, existing proposals for update languages for the Web rely on a path-expression-matching operation that selects nodes within the input document; the selected nodes are the target of the update operations. XChange follows *another approach* for updating data – update specifications are *patterns* for the data to be updated augmented with the desired update operations. By following a pattern-based approach for XChange updates,

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

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

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

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

Res_Spec ::= Uri ("," Format)?
```

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

The outline of this section is as follows: Section 4.6.1 introduces the notion of *update term* and discusses general characteristics of update operations in XChange. The sections that follow dicuss in more detail the three kinds of update operations offered by XChange. The discussion on insertion (Section 4.6.2), deletion (Section 4.6.3), and replacement (Section 4.6.4) specifications applies to updating subterms of data terms; the special case of updating the root of a data term is discussed in Section 4.6.5. Discussions on the chosen approaches to updating data are offered throughout the whole section.

4.6.1 Update Terms

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

Definition 4.1 (Update Term)
An *update term* is an Xcerpt query term augmented with the desired update operations. The query term gives a pattern for the data to be modified. The update operations may be insertions, deletions, or replacements of data. An *insertion operation* specifies an Xcerpt construct term that is to be inserted, a *deletion operation* specifies an Xcerpt query term for deleting all data terms matching it, and a *replace operation* specifies an Xcerpt query term to determine data terms to be modified and an Xcerpt construct term for their new values.

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

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

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

Label   ::= label | "var" Var_Name
UQ_List ::= ((Query_Term | U_Term) ",")* U_Term ("," (Query_Term | U_Term) )*
        | U_Op
U_Op    ::= Ins_Op
        | Del_Op
        | Rep_Op
```

Lifting (removing) all update operations from an update term produces a query term. This query term will be called in the following the *subjacent* query term of the corresponding update term. For obtaining an update term, update operations are specified in a (subjacent) query term wherever a query subterm can be specified. To reiterate, an update term is made of a subjacent query term and update operations for Web resources to be modified. A discussion of the roles these two components have follows. The subjacent query term of an update term

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

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

The update operations specified in an update term

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

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

(c) use variables bound in the subjacent query term, the event part (event query), the condition part (Xcerpt query) of the XChange rule whose action contains the updates to be executed.

4.6.2 Insertion Specification

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

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

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

Order_Ins  :: =  "after"  Query_Term    "insert"  Construct_Term
           | "insert"  Construct_Term   "before"  Query_Term
           | "at"  Position  "insert"   Construct_Term

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

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

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

Example 4.38 (Simple XChange Update Term Specifying Insertion (1))
The following update term specifies insertion of data term $d\{e\{\}, f\{\}\}$ in a data term that matches the subjacent query term $a\{\{b\{\{\}\}, h\{\{\}\}\}\}$.

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

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

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

   Before update                         After update
```

Variables can be used inside insertion specifications; they are bound in the subjacent query, the event query, and/or the Xcerpt query of a rule and used for constructing the data to be inserted. The bindings for the variables are nondeterministically chosen from the set of variable substitutions resulting from evaluating the subjacent query term and the other parts of the associated rule. Recall that Xcerpt construct terms may contain grouping constructs (`all`, `some`) for gathering *all* or *some* of the variable bindings.

Example 4.39 (Simple XChange Update Terms Specifying Insertion (2))
The following update term specifies insertion of a data term that is constructed from $d\{var\ X\}$ in a data term that matches the subjacent query term $a\{\{var\ X\}\}$. The variable binding that is used to construct the data term to be inserted is nondeterministically chosen from the set of bindings that results from matching the subjacent query term with the data term to be modified.

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

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

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

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

```
  a {{                                a {
     var X,                              b { i{}, j{} },
     insert d { all var X }              h { k{} },
  }}                                     d {
                                              h { k{} },
                                              b { i{}, j{} }
                                         }
                                      }
```

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

Example 4.40 (Simple XChange Update Terms Specifying Insertion (3))
Suppose now that one needs to introduce c-labelled elements (right) after each b-labelled element child of the root (i.e. as next sibling node in the tree representation of data); assume that the data term to be modified is ordered. Consider the following update term (bindings for the variable X are obtained from the other parts of the rule having the update term as action):

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

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

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

    Before update                        After update
```

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

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

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

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

Example 4.41 (Simple XChange Update Terms Specifying Insertion (4))
The following update term inserts at last position in a catalogue a discount of 10 percent for all products not of type `New Arrival`.

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

Multiplicity Specification With the insertion constructs exemplified until now one can not make insertions of the form e.g. *insert all var Product*, as *all var Product* is not an Xcerpt construct term. But such kind of insertions are useful in practice. Moreover, one might need not only to insert all data terms constructed from a given construct term, but perhaps just some of these data terms (possibly chosen by means of a certain criteria). For this, the grouping constructs `all` and `some` of Xcerpt are used; recall that they gather *all* and *some*, respectively, possible instances of the construct term they precede. An order of the data terms to be inserted can be specified by means of the `order by` construct followed by a criteria and list of variables; the order is determined by the given criteria "applied" on the values of these variable bindings. (A detailed explanation of the grouping constructs can be found in [125], Section 4.6.2, pages 93 – 99.)

Example 4.42 (Simple XChange Update Terms Specifying Insertion (5))
At some universities (e.g. the national universities in Romania) the best students are "awarded" by receiving a studentship for the next teaching term. The following update term is used to introduce in a data term the five best students (best in terms of their final grade) per teaching unit, for the ended teaching term.

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

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

```
in { or { resource {"file:ai.xml", "file:net.xml", "file:db.xml"} },
   desc TeachingUnit [[
           Name { var N },
           var Term → TeachingTerm {{ }},
           Students [[
              var Stud → Student {{
                             FinalGrade { var Grade }
                        }}
           ]]
        ]]
}
```

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

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

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

On Introducing Other Constructs for Specifying Insertion in XChange (Discussion) A question arose during the design phase of the language XChange, namely whether a construct like

```
 insert var Var1 with child var Var2
```

is needed or not, as sometimes the structure of the data term bound to variable *Var*1 is unknown. For example, consider that one needs to insert the content of *a*-labelled subterms into *b*-labelled subterms, but with a new child element (variable *New* is bound in the other parts of the rule having the update term as action part). Having a language construct like the one introduced above, the desired update can be specified like:

```
root {{
   a {{ var C }},
   b {{
      insert var C with child var New
   }}
}}
```

The structure of the bindings for *C* is not known; though, dispensing with a new language construct, using a variable instead of the elements’ label and the optionality construct of Xcerpt, the update can be specified like:

```
root {{
   a {{ var Label {{ optional var Content }}  }},
   b {{
      insert var Label {{
                 all optional var Content,
                 var New
              }}
   }}
}}
```

Writing such kind of update terms is not that complex for having one update construct more in the language XChange; thus, the construct discussed here is not in the current version of XChange.

4.6.3 Deletion Specification

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

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

$label\{\{q_1, q_2, ..., q_n, q_{n+1}, q_{n+2}, ..., q_m\}\}$. The rational behind it is that for deleting subterms of a data term, these subterms need to exist, that is the data term to be modified needs to contain subterms matching the query terms following the `delete` keyword.

The grammar rules defining XChange deletion operations are given next:

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

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

Example 4.43 (Simple XChange Update Term Specifying Deletion (1))
Assume that a data term is to be modified that matches the query term given next on the left. All *c*-labelled subelements of *a*-labelled elements having at least one subelement labelled *b* are to be deleted. The update term given on the right specifies the deletion. *Note* how easy the desired delete operation can be specified: one only needs to put the keyword `delete` in front of the query term that matches the data terms to be deleted.

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

Example 4.44 (Simple XChange Update Term Specifying Deletion (2))
The following update term deletes all subterms labelled *b*, regardless of their depth within the *a*-labelled term representing the whole document to be modified. *Note* that only *b*-labelled subterms are deleted, their parents (more concrete, their ancestors) remain in the data term if they do not have label *b*.

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

For inserting data terms at a given position, a construct is offered in XChange; for deleting data terms having a given position and matching a given query term, no extra construct is needed. This can be specified inside the query term by means of the construct `position` of the language Xcerpt; a query term of the form *position Pos q* matches those data terms *t* found at position *Pos* in the queried data term, where $q \preceq t$. The position specification *Pos* is either a positive integer (where 1 is the position of the first subterm of a parent), a negative integer (where -1 is the position of the last subterm), or a variable that matches with the position of subterm (matching *q*) and binds to it as a positive integer (cf. [125], Section 4.3.3, pages 73-74).

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

Example 4.45 (Simple XChange Update Term Specifying Deletion (3))
The following update term deletes maximal two subterms of the root element; they match $b\{\{var X, var Y\}\}$, where the lexical order on the values of variable *X* determine which are the two terms to be deleted.

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

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

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

On Deletion Operations on Graph Data (Discussion) When dealing with graph data, more than one approaches for deleting data are conceivable; they are determined by the treatment of dangling references. When removing a subgraph *sg* from a graph *g*, two approaches can be followed: Remove also all references to the nodes in the subgraph *sg*; this kind of deletion is known as *greedy deletion*. Leave those parts of *sg* that are referenced by other nodes from the “remaining” graph *g*; this kind of deletion is known as *parsimonious deletion*. At moment, XChange deletion operations follow the greedy deletion approach. Supporting both kinds of deletions and offering means to choose the desired one is one of the perspectives for future work.

4.6.4 Replacement Specification

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

The grammar rules defining XChange replacement operations are given next:

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

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

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

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

Proposition 4.2 (Necessity of Replacement Operation)
A replace operation of the form *Q_Term replaceby C_Term* has not the same effect as the sequence of two update operations of the form *delete Q_Term* and *insert C_Term*, where *Q_Term*, *C_Term* are specifications of (possibly) multiplicity followed by a query term and a construct term, respectively.

Cf. Proposition 4.2, XChange replace operation is not syntactic sugar for a delete and an insert operations performed in sequence. The next, simple example intends to clarify the effect of a replace operation in XChange and motivates the previous statement.

Example 4.46 (Simple XChange Update Term Explaining the Replace Operation)
The example gives an update term that specifies the replacement of each binding for the variable X with a binding for the variable Y.

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 4.47 (XChange Update Term Converting Prices from Euro to Dollar)
The example gives an XChange update term that specifies the modification of the used currency from Euro to US Dollar. The prices for *all* flights offered by a specific airline are modified accordingly to an exchange rate.

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

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

On Introducing Other Constructs for Specifying Insertion and Replacement in XChange (Discussion) Language constructs are introduced for easing the programming task. As already discussed in Section 4.4, a tradeoff between the expressive power of a language and the ease of its usage needs to be found in designing a language. Decisions need to be taken for introducing (or not) new constructs when use cases are discovered that can not be solved with the existing language constructs.

Example 4.48 (Example Motivating a New Language Construct)
Assume that the XML files `catalogue-eu.xml` and `catalog-usa.xml` contain information about products a company provides on the European and American market, respectively. One wants to insert all products of class “New Arrival” found in the European catalogue into the American one; the product prices should be calculated and inserted directly in US Dollar.

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

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

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

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

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

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

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

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

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

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

4.6.5 Special Case – Updating the Root

The insertion, deletion, and replacement specifications introduced in the previous sections augment query terms for obtaining update term specifications. The premise of modifying desired data terms is that the query terms – called subjacent query terms – simulation unify with the data terms. This is the case for updating data terms that contain data (i.e. they are not empty) or the data need not be overwrite. For *updating the root* – inserting a data term into an empty Web resource (actually 'constructing' a new resource), deleting the whole data term found at a Web resource (actually deleting the resource), or overwriting (replacing) the data term at a Web resource – a subjacent query term is neither needed nor allowed, a single update operation will do.

Imagine an XChange update like a function that takes as arguments an update term and a data term, and returns a (updated) data term. Data terms represent trees (cf. Section 2.2.1), thus, an XChange update can be represented as a function that applies an update term to a tree for obtaining the updated tree. Regardless whether one wants to update the root of a tree, the whole tree, or just parts of it, the result of the update needs to be a *tree*. This requirement has consequences on the possible update operations for updating the root. Their syntax is the same as for the update operations introduced so far; though, not all update specifications are allowed for updating the root so as not to violate the previously given requirement.

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

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

Insertion as Root In XChange, the update used for inserting data (more precisely a data term) at a given Web resource *Res* has the following form

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

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

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

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

Example 4.49 (Insertion as Root)
After the next update is executed, the data term found at `http://software.de/products.xml` is found at `http://sn.de`.

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

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

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

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

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

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

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

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

Example 4.50 (Deletion of the Root)
An update that specifies the deletion of the data of `http://software.de/products.xml` (i.e., the entire XML document is to be deleted).

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

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

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

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

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

Example 4.51 (Replacement of the Root)
An update that specifies the replacement of the data term found at `http://sn.de` with the data term found at `http://software.de`.

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

The binding for the variable *P* is obtained from evaluating the 'condition part' of the rule having the above update as 'action part'; the Web query of 'condition part' looks as follows:

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

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

Following the first approach that has been investigated, the desired updates have been executed directly on bindings for variables returned from evaluating an Xcerpt query. Xcerpt queries are specified as patterns for the data and do not select nodes, but return data terms as bindings for the variables in the query – they are *copies* of data terms. Thus, performing updates *directly on* the data terms obtained as answers to Xcerpt queries means that only copies of the data are modified; such updates *do not update* (modify) the data. Recall the Example 4.47, which specifies the modification of the used currency from Euro to US Dollar. Consider now the following query term that is needed for binding data terms to variables and modify them afterwards:

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

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

```
in { resource { "http://airline.com" },
     in var Ch replace var L by var Today,
     in var Curr replace "EUR" by "Dollar",
     in var P replace var Price by var Price * var Exchange
   }
```

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

Another idea for updating persistent data is to give the intended updates "implicitly" by specifying how data should look after the updates are performed; following such an approach consists in specifying the *result* of the updates instead of the *way* (given through explicit update operations) towards the desired result. This would mean that, instead of XChange update terms, construct terms are used that give a pattern for the data after the update. For determining the update operations to be executed, a diff function can be used. This idea has not been materialised in XChange, as specifying the result of the updates is not that easy as it seemed at first glance. A small set of update operations and update constructs give rise to more simple update specifications. However, thinking in the other direction – to translate XChange update terms into deductive, construct-query rules – has yield interesting results that are given in Section 5.1.3.

4.7 Complex Updates as Transactions

XChange updates may be *elementary* or *complex*. An *elementary update* is a change (i.e. insert, delete, replace) to a persistent data item (e.g. XML or RDF data) that can be expressed by means of an update term. *Complex updates* expressing ordered or unordered conjunctions, or disjunctions of (elementary or complex) updates are also offered by XChange. Such updates are often required by real applications. E.g. when booking a trip on the Web, one might wish to book an early flight *and* the corresponding hotel reservation, *or* else a late flight *and* a shorter hotel reservation. Since it is sometimes necessary to execute such complex updates in an *all-or-nothing manner* (e.g. when booking a trip, a hotel reservation without a flight reservation is useless), XChange has a concept of transactions.

The grammar rule defining XChange updates is the following (an XChange update is an elementary or a complex update):

```
Update     ::= E_Update | C_Update
```

The next section discusses elementary updates by shortly revisiting the notion of update patterns introduced in the previous section. Complex updates are combinations of elementary and complex updates; they are introduced in Section 4.7.2. XChange transactions, i.e. XChange updates executed in an all-or-nothing manner, are discussed in Section 4.7.3.

4.7.1 Elementary Updates

An *elementary update* specification is an update term specification accompanied by a resource specification (giving the Web resources to be modified). Section 4.6 offered an introduction to XChange update terms – patterns for the data to be updated augmented with the desired update operations. The three kinds of XChange update operations (insertions, deletions, and replacements) have been described almost independent from each other (Sections 4.6.2, 4.6.3, and 4.6.4). However, more than one update operations can be specified in an update term (nesting update operations does not make sense). Update operations are specified as subterms inside a query term – called subjacent query term of the update term. Though, for updating the root of a data term (e.g. inserting data into an empty resource), no subjacent query term is needed – a single update operation is used.

Definition 4.3 (Subjacent Query Term of an Update Term)
Given an update term u, the *subjacent query term* of u (denoted sub_u) is an Xcerpt query term obtained by removing from u

(a) the insertion operations, except the query terms they may contain due to `after` or `before` position specifications;

(b) the `delete` keywords and the multiplicity specifications of delete operations;

(c) the replace operations, except the query terms they contain.

Steps *(a)*, *(b)*, and *(c)* can be performed in arbitrary order for obtaining sub_u.

For example, for insertion specifications of the form *after Query_Term insert Construct_Term*, the *after* and *insert Construct_Term* are to be removed for obtaining the subjacent query term of an update term containing the update operation. Subjacent query terms of XChange update terms play an important role in the execution of the desired updates.

Proposition 4.4 (Role of Subjacent Query Terms)
Let $U = (d, u)$ be an elementary update with data term d to be modified by update term u. If $sub_u \preceq d$ does not hold, no update operations of u are to be "applied" to d.

If the subjacent query term of an elementary update matches the data term to be modified, the specified update operations are executed. XChange update operations are *intensional updates*, they are a description of updates in terms of (standard or event) queries. They can be specified in XChange as the language inherits the querying capabilities of the language Xcerpt. This eases considerably the specification of updates, e.g. for specifying *modification* of the discounts for *all* flights offered by a specific airline.

Example 4.52 (XChange Elementary Update)
At `http://airline.com` the flight timetable needs to be updated as reaction to flight cancellations. The information about the cancelled flight is obtained from the event part of the rule having the following elementary update as action part.

```
in { resource { "http://airline.com" },
    flights {{
      last-changes { var L replaceby var RTime },
      flight {{ number { var N }, date { var RTime },
                delete departure-time {{ }},
                delete arrival-time {{ }},
                insert news { "Flight has been cancelled!!" }
       }}
     }}
   }
```

4.7.2 Complex Updates

An XChange *complex update* is an ordered or unordered, conjunction or disjunction of updates (i.e. of elementary or complex updates). A conjunction of updates expresses that all specified updates are to be executed. A disjunction of updates expresses that one of the specified updates is to be executed.

Complex updates specifying conjunctions are introduced by the keyword `and`, disjunctions by the keyword `or`. Specifications denoting that the XChange update is a complex one are always total (i.e. partial conjunctions or disjunctions of updates do not make sense). Square brackets and curly braces are used for denoting that the order of evaluation is of importance or of no importance, respectively. The grammar rules defining complex updates in XChange are given next:

```
C_Update      ::= Ordered_CU | Unordered_CU
Ordered_CU    ::= "and" "[" Update_List "]"
```

```
             | "or" "[" Update_List "]"
Unordered_CU ::=  "and" "{" Update_List "}"
             | "or" "{" Update_List "}"
Update_List  ::= Update ("," Update)+
```

XChange offers four kinds of complex updates – ordered and unordered conjunctions, ordered and unordered disjunctions of updates. The effect of such updates and the scope of variables occurring in complex updates are explained in the following for each of these kinds.

Ordered Conjunction of Updates Consider an XChange complex update specification o_conj of the form $o\_conj = and[u_1, u_2, ..., u_n]$, where $2 \leq n$ and u_i, $1 \leq i \leq n$ specify XChange updates. The *effect* of o_conj is the effect of executing all u_i ($1 \leq i \leq n$) sequentially in the order of their occurrence in the list. This means that the effect of an update u_i is "visible" for updates u_j, with $j > i$. The visibility of update effects is twofold:

(a) the bindings for the variables of u_i that are obtained by evaluating u_i can be used in the evaluation of u_j with $j \leq i+1$;

(b) consider updates u_i and u_j with $j > i$ that modify the data found at a resource *Res*, and there is no u_k with $i < k < j$ that modifies *Res*. Data term d is found at *Res* before the updates are performed. Thus u_i modifies d and results in having at *Res* a data term d_i, whereas u_j modifies the data term d_i.

Example 4.53 (XChange Complex Update Specifying Sequence of Updates)
The following XChange complex update specifies that a flight reservation and a hotel reservation are to be executed in the specified order. After giving the shape of such an update, an instantiation of it follows.

```
and [
  <make flight reservation>,
  <make hotel reservation depending on the flight schedule>
  ]
```

The following complex update specifies that a flight and a corresponding hotel reservations are to be made for Christina Smith. The bindings for the variables F (the chosen flight) and H (the chosen hotel) are obtained from the other parts of the rule having the update as action part. *Note* that the variables B and E are bound during the evaluation of the first update and used afterwards in evaluating the second update of the conjunction.

```
and [
   in { resource { "http://travel-agency.net/flights/" },
        desc reservations {{
               outward-date { var B ->"2005-08-21" },
               return-date { var E ->"2005-08-22" },
              insert reservation {
                     var F, name { "Christina Smith" } }
        }}
      },
   in { resource { "http://hotels.net/reservations/" },
        accommodation {{
          insert reservation {
                  var H, name { "Christina Smith" },
                  from { var B }, until { var E }  }
        }}
       }
  ]
```

As the example above shows, ordered conjunction of updates are amenable to applications involving sequences of updates to be executed, where the order of update execution plays an important role. They are useful when data gathered or used in an update is needed for executing subsequent ones, or when complex modifications to the same Web resources' data are needed.

Unordered Conjunction of Updates Consider an XChange complex update specification u_conj of the form $u\_conj = and\{u_1, u_2, ..., u_n\}$, where $2 \leq n$ and u_i, $1 \leq i \leq n$ specify XChange updates. The *effect* of u_conj is the effect of executing all u_i, $1 \leq i \leq n$, in some arbitrary execution order. The order of their execution is not given and, thus, the runtime system has the freedom to choose the execution order.

The *scope* of variables used in update u_i is restricted to u_i, i.e. the bindings for the variables resulted from evaluating u_i can not be used in the evaluation of u_j with $i \neq j$. (This restriction can be lifted, for parallel evaluation of updates; variable substitutions need to be communicated between the Web sites where the data to be modified are found.) Unordered conjunction of updates are suitable for specifying updates to be executed that do not "share" other variable bindings than the ones received from the event query and/or Web query of the rules whose action they represent.

Note that unordered conjunction of updates that modify the same data may have different results, depending on the order of their evaluation. This is illustrated by means of an example.

Example 4.54 (XChange Complex Update Specifying Unordered Conjunction of Updates)
The following example specifies that a deletion and an insertion should be executed on the same data, in `test.xml`.

```
and {
  in { resource { "file:/test.xml" },
       a {{ delete b {{}} }},
  },
  in { resource { "file:/test.xml" },
       a {{ insert b { f { "content"} } }},
  }
}
```

Consider that `test.xml` contains the following data term before any of the updates specified above are executed:

```
a {
  b { f { "info" } },
  b { g { "info" } },
  c { h { "info" } }
}
```

The possible results of executing the unordered conjunction of updates on `test.xml` are given next. Depending on the order in which the two updates are executed, one of the following data terms are obtained (on the left hand side the result of executing the deletion followed by the insertion, on the right hand side the result of insertion followed by the deletion):

```
 a {                                          a {
   b { f { "content" } },                       c { h { "info" } }
   c { h { "info" } }                         }
 }
    Delete, insert                            Insert, delete
```

Ordered Disjunction of Updates Consider an XChange complex update specification o_disj of the form $o\_disj = or[u_1, u_2, ..., u_n]$, where $2 \leq n$ and u_i, $1 \leq i \leq n$ specify XChange updates. The *effect* of o_disj is the effect of executing one single u_i, $1 \leq i \leq n$; the disjunction of updates is an exclusive one. The ordered specification expresses that the runtime system should try to execute the updates in the given order, until a (first) update has been successfully executed. Like for unordered conjunctions of updates, the *scope* of variables used in update u_i is restricted to u_i. Since just one update is to be executed, one can not discuss visibility of updates' effects.

Example 4.55 (XChange Complex Update Specifying Disjunction of Updates)
The following XChange complex update specifies that a travel reservation is to be performed, if no flight reservation can be made, a train ticket should be reserved. The disjunction update has the following shape:

```
or [
  <make flight reservation>,
  <reserve train ticket>
  ]
```

The above template is instantiated to make the desired reservation for Christina Smith; bindings for the variables are obtained from evaluating the event query and Web query of the rule having the update as action.

```
or [
  in { resource { "http://lhs.de/flights/" },
        desc reservations {{
          insert reservation {
                  var Flight, name { "Christina Smith" }  }
        }}
      },
  in { resource { "http://db.de/trains/" },
        desc tickets {{
          insert reservation {
                  var Train,  name { "Christina Smith" }  }
        }}
       }
  ]
```

Unordered Disjunction of Updates Consider an XChange complex update specification u_disj of the form $u\_disj = or\{u_1, u_2, ..., u_n\}$, where $2 \leq n$ and u_i, $1 \leq i \leq n$ specify XChange updates. The *effect* of u_disj is the effect of executing one single u_i, $1 \leq i \leq n$. The runtime system has the freedom to choose the order in which it tries to find and successfully execute one of the updates. Like for ordered disjunction of updates, the *scope* of variables used in update u_i is restricted to u_i.

4.7.3 Transactions

An XChange *transaction specification* is a group of (elementary or complex) update specifications and/or explicit event specifications (expressing events that are constructed, raised, and sent as event messages) that are to be executed in an *all-or-nothing manner*. That is, either all specified actions are successfully executed or none of the updates are executed (partial effects of the updates need to be undone).

Elementary and complex update specifications have been introduced in the previous sections. They specify (local or remote) Web resources to be modified and the updates to be performed on their data. An XChange *event specification* is a (complete) *pattern* for the event message(s) to be constructed and sent to one or more Web sites. The notion of *event terms* is used to denote such patterns for events to be raised. An event term represents a restricted construct term that may be preceded by the keyword `all`.

A restricted construct term is an Xcerpt construct term having root labelled `XChange-Namespace:event` and at least one subterm `XChange-Namespace:recipient`$\{uri\}$ that specifies a Web site's address. The constructed event message is to be sent to the XChange program found at *uri*. If more than one subterms of the form `XChange-Namespace:recipient` $\{uri\}$ are given in an event term, the constructed event message is to be sent to all specified recipient Web sites.

An event term of the form `all Construct Term` is used to raise and send *all* events that are constructed with `Construct Term` by applying the substitutions obtained from the rest of the XChange reactive rule whose head specifies the event term. Such event terms are useful e.g. when the event messages to be sent have different content (depending on the variable substitutions). Examples can be found in Section 6.1, where application scenarios for XChange are given.

Actually, *Event_Term* in the grammar rules given next is not a construct term with arbitrary structure – it has been generalised to construct term for reasons of simplicity. The following grammar rules define transactions specifications in XChange:

```
Trans      ::= Update
           | Event_Term
           | Ordered_AList
           | Unordered_AList
Event_Term       ::= Construct_Term
                 | "all" Construct_Term
Ordered_AList    ::= "and" "[" AList "]"
                 | "or" "[" AList "]"
Unordered_AList ::=  "and" "{" AList "}"
                 | "or" "{" AList "}"
AList  ::= Update ("," Action)+
Action ::= Update | Event_Term
```

A transaction specification can be considered as an ordered or unordered conjunction or disjunction of *action* specifications. At moment, updates and event terms are considered as actions in XChange. However, this view offers flexibility in extending XChange with other kind of actions if considered necessary. The discussion on visibility of update effects for other updates inside a transaction (including also the usage of obtained variable substitutions) can be ported to the more general setting of actions (and covering thus event terms or combinations of updates and event terms).

On XChange Transactions and Their Management on the Web Combinations of XChange actions are considered transactions if they obey the ACID properties [136] (Atomicity, Consistency, Isolation, and Durability). Section 1.3.6 has explained each of the properties a transaction should have. Communicating transaction requests and synchronising the actions to be taken can be implemented to some extent by means of XChange rules; however, transaction-related issues deserve more investigation in the framework of XChange so as to realise transaction management on the Web. The idea is to extend XChange with standard solutions from database systems that are to be adapted to the biggest existing distributed system – the Web.

Actions performed inside of a distributed transaction on the Web may trigger local or remote actions that in turn can trigger other actions (i.e. *cascading triggering* on the Web). Upon abort of a triggering transaction, rollback of all triggered actions need to be assured on the Web, a decentralised environment posing new challenges. A nested transaction model has been proposed in [141] (for HiPac, pages 184 - 186) for accommodating with the relationship between a transaction and the rules triggered by it (which in turn can trigger other rules). How to cope with these kinds of problems has been discussed also in [141] (for Chimera, page 167) and [119]. Existing proposals for management of triggering transactions and triggered ones in database systems might prove very useful in extending XChange.

Transactions defined at the level of an XChange-aware Web site, or more concrete in an XChange program, should recognise contradictory transactions and update specifications, possibly at compile time, or develop transaction inconsistency resolution strategies. For example, for *insert a* and *delete a*, conceivable strategies would be to execute the update associated with the reactive rule with higher priority, or not to execute these two updates at all. At moment, however, XChange does not consider priorities for rules but, at the same time, leaves room for such kind of extensions.

Contingency mechanisms could be also employed for transaction management on the Web, i.e. use the events expressing abort of a transaction to specify how to react in case that a transaction aborts.

The emphasis in this thesis is not on a language for distributed transactions on the Web; the thesis recognises the need for transactions through developed application scenarios and the components a transaction on the Web might have, and proposes a syntax for (event-driven) transactions. A complete investigation and realisation (including formal semantics and implementation) of transactions on the Web are outside the scope of this thesis.

4.8 Rules

An XChange program is located at one Web site and consists of one or more rules: Reactive rules of the form *Event query – Web query – Action* specify situations of interest and the actions to be automatically executed if such situations occur. Deductive rules are Xcerpt rules that infer new data from existing (persistent) Web data (are views over persistent data).

The language XChange has been deliberately designed to mirror the clear separation between *volatile* and *persistent* data. The language constructs deal either with volatile or with persistent data for easing their understanding and usage. There are two levels of the language that mirror the view over the Web data: (a) rules' level, and (b) reactive rule components' level.

(a) XChange reactive rules specify reactions to be executed in response to incoming volatile data. In contrast, XChange deductive rules deal only with persistent data, they query persistent data and construct new persistent data. XChange deductive rules are rules expressed in the query language Xcerpt (integrated into XChange), which has been described in [125]. (Thus, the focus of this thesis is on XChange reactive rules.)

(b) Regarding the components of XChange reactive rules, the *Event query* refers to (queries) volatile data and the *Web query* refers to (queries) persistent data. The *Action* might refer to volatile data (by sending event data) or to persistent data (by updating persistent data). Rule components communicate only through variable substitutions. Substitutions obtained by evaluating the event query can be used in the Web query and the action part, those obtained by evaluating the Web query can be used in the action part.

There are two kinds of XChange reactive rules that differ in the action to be executed: *Event-raising rules* specify explicit events to be constructed and sent to one or more Web sites. *Transaction rules* specify transactions to be executed. Thus, the grammar rules defining XChange rules are the following:

```
XCRule     ::= React_Rule | Xcerpt_Rule
React_Rule ::= Raise_Rule | Trans_rule
```

XChange rules can be defined by programmers, system administrators, and non-programmers with a level of knowledge depending on their application requirements. Being a high-level language, XChange should not be to difficult to use even by non-experienced programmers. Moreover, a visual counterpart of XChange is planned that could increase the accessibility of the language. XChange rules can be defined also by applications – e.g. rules can be automatically generated based on the dependencies between Web resources' data.

This section is structured as follows: Section 4.8.1 discusses XChange event-raising rules. Section 4.8.2 focuses on XChange transaction rules. Deductive rules in XChange are motivated through an example in Section 4.8.3. This section ends by defining the range restriction of XChange rules (Section 4.8.4).

4.8.1 Event-Raising Rules

XChange *event-raising rules* are means for notifying reactive (XChange-aware) Web sites of (atomic or composite) events that have occurred. They specify event messages to be constructed and sent to other Web sites as reaction to (local or remote) events. Conditions that need to hold for raising events can be specified by means of queries to persistent data; these conditions select data items from persistent data that are used for constructing event messages.

Event-raising rules are introduced by the keyword `RAISE` followed by an event term, the (atomic or composite) event query is preceded by the keyword `ON`, and the Web query by the keyword `FROM`. Event term specifications have been introduced in Section 4.7.3. Event queries have been discussed in Section 4.4. Web queries are Xcerpt queries; a short introduction has been given in Section 2.4.2, for a more detailed discussion on Xcerpt querying capabilities see [125]. The grammar rule defining event-raising rules is given next. (*Note* that just the event term specification is mandatory in XChange event-raising rules; the event query and/or the Web query can be left out.)

```
Raise_Rule ::= "RAISE" Event_Term ("ON" EvQ)? ("FROM" Query)? "END"
```

Incoming events are queried by means of event query *EvQ*. For each answer to *EvQ* the Xcerpt query *Query* is evaluated. If *Query* evaluates successfully, an event message is constructed from *Event_Term* and is sent to the specified recipient. *EvQ* and *Query* select data from incoming events (volatile data) and Web resources (persistent data), respectively, as bindings for the variables occurring in the queries. Assuming that the answers to *EvQ* and *Query* contain the substitution sets Σ_{EvQ} and Σ_{Query}, respectively, for constructing the event message substitution set $\Sigma = \Sigma_{EvQ} \bowtie \Sigma_{Query}$ will be used.

An event that has been raised at a Web site (i.e. its representation has been constructed as event message to be sent to one or more Web sites), contains as parameters the *recipient* Web site (that needs to be given in the event term specification), the *sender* Web site, and the *raising time* (the last two are determined by the event manager of the Web site sending the event message; the event manager provides this information by inserting it into the event representation before its sending). The *reception time* and the event *id* parameters are determined and inserted by the event manager of the recipient Web site when the event message is received.

Example 4.56 (XChange Event-Raising Rule)
The site `http://airline.com` has been told to notify Mrs. Smith's travel organiser of delays or cancellations of flights she travels with. The shape of such an event-raising rule is given followed by a concrete, complete XChange event-raising rule.

```
RAISE
   <event message pattern notifying flight cancellation>
ON
   <event query detecting flight cancellations>
END
```

The following event-raising rule is registered at `http://airline.com` and detects cancellation notification events sent by one of the airline's control points. If the flight *AI*2021 is cancelled, the airline notifies the organiser of Mrs. Smith about this event.

```
RAISE
  xchange:event {
     xchange:recipient { "http://organiser.com/Smith" },
     cancellation-notification { var F }
  }
ON
  xchange:event {{
     xchange:sender { "http://airline.com/control-point20/" },
     cancellation {{
        var F -> flight {{ number { "AI2021" },
                           date { "2005-08-21" }
                  }}
     }}
  }}
END
```

4.8.2 Transaction Rules

XChange *transaction rules* are means for updating persistent data on the Web and notifying other XChange-aware Web sites of events occurring during the execution of these updates. These actions – updates and raising of events – are to be executed as a transaction. Conditions that need to hold, as a precondition for transaction execution, can be specified by means of queries to persistent data.

Transaction rules are introduced by the keyword `TRANSACTION` followed by a transaction specification, the (atomic or composite) event query is preceded by the keyword `ON`, and the Web query by the keyword `FROM`. Transaction specifications have been introduced in Section 4.7.3. The grammar rule defining transaction rules is given next. (*Note* that just the transaction specification is mandatory in XChange transaction rules.)

```
Trans_Rule ::= "TRANSACTION" Trans ("ON" EvQ)? ("FROM" Query)? "END"
```

As for event-raising rules, incoming events are queried by means of event query *EvQ*. For each answer to *EvQ* the Xcerpt query *Query* is evaluated. If *Query* evaluates successfully, the actions specified in transaction *Trans* are to be executed (either all of them or none). Assuming that the answers to *EvQ* and *Query* contain the substitution sets Σ_{EvQ} and Σ_{Query}, respectively, for executing the specified actions (updates and events to be raised) substitution set $\Sigma = \Sigma_{EvQ} \bowtie \Sigma_{Query}$ will be used.

Note that the 'event part' is not mandatory for event-raising rules and transaction rules in XChange, so as to be able to specify e.g. updates that are to be executed not (necessarily) as reaction to events.

Example 4.57 (XChange Rule for Booking a Flight)
The travel organiser of Mrs. Smith uses the following rule: if the return flight of Mrs. Smith is cancelled then look for and book another suitable flight. Again, the shape of such a transaction rule is given first.

```
TRANSACTION
   <make flight reservation>
ON
   <event query detecting flight cancellations notifications>
FROM
   <Web query looking for another suitable flight>
END
```

The XChange rule of Example 4.56 is used to send event messages to Mrs. Smith's organiser; the next XChange transaction rule responds to it by booking another flight. If no suitable flight is found, no action is performed.

```
TRANSACTION
  in { resource { "http://airline.com/reservations/" },
       reservations {{
         insert reservation { var F, name { "Christina Smith" } }
        }}
     }
ON
  xchange:event {{
     xchange:sender { "http://airline.com" },
     cancellation-notification {{
        flight {{ number { "AI2021" },
                  date { "2005-08-21" }  }}
          }}
       }}
FROM
  in { resource { "http://airline.com" },
       flights {{
          var F -> flight {{
                     from { "Paris" }, to { "Munich" },
                     date { "2005-08-21" }
       }}
    }}
   }
END
```

Example 4.58 (XChange Rule Specifying Sequence of Updates as Action)
If no other suitable return flight is found and the airline does not provide an accomodation, then book for Mrs. Smith a cheap hotel and inform her secretary about the changes of her schedule. This is represented in XChange as a rule the travel organiser of Mrs. Smith has. The rule is shaped as follows:

```
TRANSACTION
   <make hotel reservation>
            <and>
   <announce secretary of changes of schedule>
ON
   <event query detecting cancellations of flights for which
    the airline does not provide an accommodation>
FROM
   <Web query looking for a suitable hotel>
END
```

For a cancelled return flight from Paris to Munich, the travel organiser of Mrs. Smith uses the following XChange rule:

```
TRANSACTION
  and [
    in { resource { "http://hotels.net/reservations/" },
         reservations {{
            insert reservation {
                    var H, name { "Christina Smith" },
                    from { "2005-08-21" }, until { "2005-08-22" } }
         }}   },
     in { resource { "diary://diary/my-secretary" },
          diary {{
            news {{
              insert my-hotel {
                       remark { "I'm staying in Paris over night!" },
                       phone { var Tel },  reason { "Flight cancellation." } }
      }}  }}   }
   ]
ON
  andthen [
    xchange:event {{
       xchange:sender { "http://airline.com" },
       cancellation-notification {{
          flight {{ number { "AI2021" }, date { "2005-08-21" }  }}
       }}
    }},
    without { xchange:event {{
                 xchange:sender { "http://airline.com" },
                 accomodation-granted {{
                     hotel {{ }} }}  }}
            } during [2005-08-21T15:00:00..2005-08-21T19:00:00]
  ] within 2 hour
FROM
   result [[
       var H -> Position 1 hotel {{ phone { var Tel } }} }}
     ]]

END
```

Note that the Web query (introduced by `FROM`) does not query a particular Web resource; it queries a view over the data of two Web resources having different structures. The Xcerpt rule constructing the view over hotel data is given in the next section (Example 4.59). Variable *H* is to be bound to the hotel offering the best price.

4.8.3 Deductive Rules

Deductive rules are means for constructing views over heterogeneous data sources. As exemplified in the previous section, data views are easily and elegantly queried in the *Web Query* part of reactive rules. Deductive rules are expressed by using the Web and Semantic Web query language Xcerpt, which is integrated into XChange. A short introduction to Xcerpt has been given in Section 2.4.2, for a more detailed discussion on Xcerpt querying capabilities see [125].

```
Xcerpt_Rule ::= "CONSTRUCT" Construct_Term "FROM" Query "END"
```

Deductive rules of an XChange program can be chained, that is query parts of reactive or deductive rules can query the result of other deductive rules. This is realised by matching (simulation unifying) the query part with the construct part – the head – of other deductive rules. *Note* that the head of reactive rules can not be queried.

Example 4.59 (Deductive Rule for Gathering Information about Hotels)
The following Xcerpt rule queries data found at Web resources `http://hotels.net` and `http://hotels-paris.fr` and constructs a view over the hotel data by gathering information about all hotels in Paris. The constructed data term contains a list of hotels ordered by their price per room.

```
CONSTRUCT
 result [
    all hotel { name { var Name },
                price { var Price },
                phone { var Phone } } order by ascending [ var Price ]
   ]
FROM
 or {
    in { resource { "http://hotels.net" },
         accommodation {{
            hotels {{
               city { "Paris" },
               desc hotel {{
                       name { var Name }
                       price-per-room { var Price },
                       phone { var Phone } }} }}
         }}
     },
     in { resource {"http://hotels-paris.fr"},
          logement {{
             hotel {{
                nom { var Name },
                telephone { var Phone },
                prix { var Price }
             }}
           }}
     }
 }
END
```

Note that the two data terms queried for hotels in Paris have different structures; the above given rule not only gathers the desired information, but also gives data a uniform structure.

Complex applications specifying reactivity on the Web require a number of features that can not always be specified by simple programs. In XChange, rules are also *means for structuring* complex XChange programs.

4.8.4 Range Restriction

This section discusses *range restriction* of XChange rules, i.e. a syntactic restriction on admissible XChange rules. The satisfaction of the range restriction property by the rules of an XChange program is assumed in defining the formal semantics of XChange (Section 5). Moreover, range restriction of XChange rules is a syntactical property that can be statically verified when parsing XChange programs so as to avoid (some) programming mistakes.

Intuitively, an XChange rule is range restricted if every variable occurring in the construct term(s) of the rule's head ('action part' or construct part) has at least one defining occurrence (i.e. an occurrence that "provides" bindings for the variable) in other parts (event query part, Web query part, or actions that are to be performed before the action containing the respective construct term) of the rule.

For defining the range restriction of XChange rules, each variable occurrence in XChange rules is associated with a *polarity* and an *optionality*; these determine whether a variable occurring in a part of the rule can be used in other parts of the respective rule. A *negative* polarity (denoted -) of a variable occurrence expresses a defining occurrence of the variable. A *positive* polarity (denoted +) expresses a non-defining variable occurrence. Optionality is given by an attribute *optional* (denoted ?) and *not optional* (denoted !) for variables contained in an optional subtree and do not always have bindings.

An XChange program is a set of rules, denoted $P = \{Rr_1, \ldots, Rr_m, Tr_1, \ldots, Tr_n, Dr_1, \ldots, Dr_p\}$, where

- Rr_i, $0 \leq i \leq m$, are event-raising rules of the form $t^e \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (t^e is an event term, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Tr_j, $0 \leq j \leq n$ are transaction rules of the form $tra \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (tra is a transaction specification, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Dr_k, $0 \leq k \leq p$ are Xcerpt rules of the form $t^c \leftarrow \mathcal{Q}$ (t^c is a construct term and $\mathcal{Q}$ an Xcerpt query), and
- $1 \leq m+n$.

The range restriction of Xcerpt rules (deductive rules Dr_k, $0 \leq k \leq p$) is defined in [125], Chapter 6, pages 133-137. Thus, this section discusses only range restriction for XChange reactive rules of a program P, i.e. event-raising rules Rr_i, $0 \leq i \leq m$, and transaction rules Tr_j, $0 \leq j \leq n$. For this, the polarity of event queries, Web queries, and actions need to be defined. As Xcerpt query terms are needed for defining the polarity of event queries and updates, and construct terms are needed for defining the polarity of event terms and updates, the definition of polarities of Xcerpt subterms is given in the following; the polarity of Xcerpt query and construct terms has been defined in [125], Chapter 6.

Definition 4.5 (Polarity of Xcerpt Subterms)

1. Let t be a query term with polarity p and optionality o.
   - if t is of the form `without` t', then t' is of polarity $+$ (regardless of p) and optionality o
   - if t is of the form `optional` t', then t' is of polarity p and optionality ?.
   - if t is of one of the forms $l\{\{t'_1,\ldots,t'_n\}\}$, $l\{t'_1,\ldots,t'_n\}$, $l[[t'_1,\ldots,t'_n]]$ or $l[t'_1,\ldots,t'_n]$ ($n \geq 0$), then t'_1, …, t'_n are of polarity p and optionality o.
   - if t is of the form `desc` t' then t' is of polarity p and optionality o.
   - if t is of the form $var\ X \rightarrow t'$ then t' is of polarity p and optionality o.
2. Let t be a construct or data term with polarity p and optionality o.
   - if t is of the form `optional` t', then t' is of polarity p and optionality ?.
   - if t is of one of the forms $f\{t'_1,\ldots,t'_n\}$ or $f[t'_1,\ldots,t'_n]$ ($n \geq 0$), then t'_1, …, t'_n are of polarity p and optionality o.
   - if t is of the forms `all` t' or `some` t', then t' is of polarity p and optionality o.

- if t is of the form $op(t'_1, \ldots, t'_n)$, with op a function or aggregation identifier, then $t'_1, \ldots, t'_n$ are of polarity p and optionality o.

The root of a query term is usually of negative polarity (and thus define variable bindings) and not optional. The root of a construct or data term is of positive polarity and not optional.

Polarity of Event Queries

For attributing a polarity for each occurrence of a variable in an XChange event query, polarities are recursively attributed for each component of an event query. An XChange event query may be atomic or composite; an atomic event query is an Xcerpt query term with an optional absolute temporal restriction specification. The above given definition (taken from [125]) represents the base of defining the polarity of XChange event queries; it defines polarity of atomic event queries without temporal restrictions. The next definition is used for defining the polarity of XChange event queries.

Definition 4.6 (Polarity of XChange Event Queries)
Let eq be an event query with polarity p and optionality o. If eq is of the form:

- eq is an Xcerpt query term, then the Definition 4.5 is used for attributing polarity to its subterms;
- $eq = eq'$ `in` $[b..e]$, or $eq = eq'$ `before` e, then eq' is of polarity p;
- $eq = eq'$ `within` w, then eq' is of polarity p;
- $eq = \texttt{and}\{eq_1, \ldots\ eq_n\}$, then eq_i is of polarity p, $1 \leq i \leq n$;
- $eq = \texttt{andthen}[eq_1, \ldots, eq_n]$ or $eq = \texttt{andthen}[[eq_1, \ldots, eq_n]]$, then eq_i is of polarity p, $1 \leq i \leq n$;
- $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]]$, then eq_i and q_{ij} are of polarity p, $1 \leq i \leq n$, $j = i+1$, $2 \leq j \leq n$;
- $eq = \texttt{or}\{eq_1, \ldots\ eq_n\}$, then eq_i is of polarity p, $1 \leq i \leq n$;
- $eq = \texttt{var}\ X \rightarrow eq'$, then eq' is of polarity p;
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$, then eq_1 is of polarity + (regardless of p) and eq_2 is of polarity p;
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ [b..e]$, then eq_1 is of polarity + (regardless of p);
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ \{eq''\}$, then eq' and eq'' are of polarity p;
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ [b..e]$, then eq' is of polarity p;
- $eq = \texttt{every}\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\ \}$, then eq' is of polarity p;
- $eq = \texttt{withrank}\ n\ \texttt{any}\ \texttt{var}\ X_1, \ldots, \texttt{var}\ X_k\ \{eq'\ \}$, then eq' is of polarity p;
- $eq = \texttt{last}\ \{eq_1\ \}\ \texttt{during}\ \{eq_2\ \}$, then eq_1 and eq_2 are of polarity p;
- $eq = \texttt{last}\ \{eq_1\ \}\ \texttt{during}\ [b..e]$, then eq_1 is of polarity p;
- $eq = (\texttt{atleast}|\texttt{atmost})?\ m\ \texttt{of}\ \texttt{any}\ \texttt{var}\ X_1, \ldots \texttt{var}\ X_k\ \{eq_1, \ldots eq_n\}\ \texttt{during}\ \{eq'\}$, then eq' and eq_i are of polarity p, $1 \leq i \leq n$;
- $eq = (\texttt{atleast}|\texttt{atmost})?\ m\ \texttt{of}\ \texttt{any}\ \texttt{var}\ X_1, \ldots \texttt{var}\ X_k\ \{eq_1, \ldots, eq_n\}\ \texttt{during}\ [b..e]$, then eq_i is of polarity p, $1 \leq i \leq n$.

Each of the component event queries of the event query eq having one of the forms given above are of optionality o.

The root of an event query is of negative polarity (it defines variable bindings) and not optional. If event exclusion is specified, the polarity changes according to Definition 4.6.

Polarity of Web Queries

Web queries in XChange event-raising rules or transaction rules are Xcerpt queries. The polarity of Xcerpt queries has been defined in [125], Chapter 6, Definition 6.2.

Polarity of Actions

XChange actions are raising of events to one or more reactive Web sites, or executing XChange transactions, i.e. ordered or unordered conjunctions or disjunctions of (elementary or complex) updates and/or raising of events.

Definition 4.7 (Polarity of XChange Event Terms)
Let *et* be an event term with polarity p and optionality o.

- if *et* is an Xcerpt construct term, then the Definition 4.5 is used for attributing polarity and optionality to its subterms;
- if *et* is of the form `all` *ct*, with *ct* Xcerpt construct term, then *ct* is of polarity p and optionality o.

An XChange update term is a pattern for the data to be updated augmented with update operations. For attributing polarity to update terms, it suffices to attribute polarity to their *subjacent query terms* and the *construct terms* of the update operations (insertion and replacements). (Recall that XChange update operations can not be nested.) The subjacent query term of an update term (Definition 4.3 in Section 4.7) is an Xcerpt query term; thus, for attributing polarity to the subjacent query terms, the part for query terms of Definition 4.5 is used. For attributing polarity to construct terms that are part of update operations (e.g. `insert` *ct* or *qt* `replaceby` *ct*, with *ct* construct term and *qt* query term), the part for construct terms of Definition 4.5 is used.

The root of a subjacent query term is of negative polarity (it provides bindings) and not optional; the root of a construct term that is part of an update operation is of positive polarity (it consumes bindings) and not optional.

Definition 4.8 (Polarity of XChange Actions)
Let a be an XChange action specification of polarity p.

1. If a is an event term, then the Definition 4.7 is used for attributing polarity to its subterms.
2. If a is an elementary update, then its subjacent query term is of polarity p and its construct terms of polarity +; the specified resources are of polarity +.
3. If a is of one of the forms:
   - `and` $[a_1, a_2, \ldots, a_n]$ or `and` $\{a_1, a_2, \ldots, a_n\}$, then every update a_i is of polarity p and every event term a_j of polarity +, $1 \leq i \leq n$, $1 \leq j \leq n$;
   - `or` $[a_1, a_2, \ldots, a_n]$ or `or` $\{a_1, a_2, \ldots, a_n\}$, then every update a_i is of polarity p and every event term a_j of polarity +, $1 \leq i \leq n$, $1 \leq j \leq n$.

The root of a complex update is of negative polarity and not optional.

Example 4.60 (Polarity in an XChange Rule)
The following event-raising rule is used for settling an appointment with one of Mrs. Smith's friends. As all terms have associated attribute not optional, the optionality is not depicted in the example.

```
RAISE
  +xchange:event {
      +xchange:recipient {"http://organiser.com/Eva"},
      +proposal {
         +text {"Can we meet?"},
         +on { +var Date }, +at {"14:00"}
```

```
       }
   }
ON
   -and {
      -without {
          +xchange:event {{
             +meeting {{
                +begin { +var Time },
                +date  { +var Date }
             }}
          }}
      } during [2005-08-21..2005-08-22],
      -xchange:event {{
         -xchange:sender {"http://organiser.com/Eva"},
         -info {{
            -text {"I'm in Munich"},
            -date { -var Date }
         }}
      }}
   } before 2005-08-22T22:00
FROM
   -in { +resource { "file:/appointments.xml" },
        -schedule {{
           -desc -appointments {{
                   -without +appointment {{
                                     +for { +var Date } }}
                }}
        }}
   }
END
```

Note that variable `Time` occurs only once and with positive polarity, that is `Time` has no occurrence that provides bindings for it. The variable `Date` occurs once with negative polarity, i.e. the event query against notifications from Eva provides bindings for `Date`; the variable has more than one occurrence with positive polarity expressing occurrences where the bindings for `Date` are consumed (e.g. in the event term proposing an appointment).

Range Restriction of XChange Rules

An XChange transaction rule is range restricted if variables occurring in the construct terms of the 'action part' have at least one defining occurrence in the event query, Web query, or in the updates that are to be performed before the respective variables are used. An XChange event-raising rule is range restricted if variables occurring in the event term have at least one defining occurrence in the event query or Web query.

For simplifying the definition of range restriction of XChange rules, the disjunctive normal form of an XChange rule is defined next.

Definition 4.9 (Disjunctive Rule Normal Form of XChange Rules)
An XChange rule $A \leftarrow_r Q \leftarrow_r Eq$ is in disjunctive normal form if Q and A are in disjunctive normal form, that is

(i) an Xcerpt query Q is in disjunctive normal form if it is of the form $\bigvee_j q_j$, where q_j is a conjunction of query terms and/or negated query terms;

(ii) an action A is in disjunctive normal form if it is an event term or a disjunction of actions, where each action is a conjunction of elementary updates and/or event terms; the conjunction of actions preserves the order of action specifications.

Bringing a Web query in its disjunctive normal form is rather straightforward and has been discussed in [125], Chapter 6. The steps followed for transforming an XChange action specification into its disjunctive normal deserve some explanation. For bringing an action specification A in its disjunctive normal form,

1. each *ordered disjunction* of A is transformed into an unordered one; this step does not influence the bindings of the variables occurring in the actions specified in the disjunction (recall that component actions of an unordered or ordered disjunction "share" only the variable bindings obtained from evaluating the event query and the Web query). Thus, actions of the form $\texttt{or}[a_1, a_2, ..., a_n]$ are transformed into $\texttt{or}\{a_1, a_2, ..., a_n\}$.

2. each *unordered conjunction* of action specification of A is transformed into a disjunction of ordered conjunctions. For example, an unordered conjunction of the form $\texttt{and}\{a_1, a_2\}$ is transformed into

   $\texttt{or}\{$

   $\quad\texttt{and}[a_1, a_2],$

   $\quad\texttt{and}[a_2, a_1] \quad \}$

   Note that the transformation preserves the definition of the scope of variables in unordered conjunction of actions (cf. Section 4.7.2).

3. the last step in the transformation consists in placing the action specification resulted from applying 1. and 2. to A into *disjunctive normal form*; this is achieved by recursively moving conjunctions inward and disjunctions outward by using the rewriting rules:

   $\texttt{and}[a, \texttt{or}\{b, c\}] = \texttt{or}\{\texttt{and}[a, b], \texttt{and}[a, c]\}$

   $\texttt{and}[\texttt{or}\{a, b\}, c] = \texttt{or}\{\texttt{and}[a, c], \texttt{and}[b, c]\}$

   By applying the above stated rules, the order of actions in ordered conjunction specifications needs to be preserved.

For defining the range restriction of XChange rules, a predicate $\texttt{prob}(eq, V)$ is defined for expressing that the event query eq provides bindings for the variable V occurring in eq. If $\texttt{prob}(eq, V) = True$, V can be consumed in other parts of the rule.

Definition 4.10 (Predicate `prob` - Provides Bindings)
Let eq be an event query and V a variable occurring in eq. The predicate $\texttt{prob}(eq, V)$ is defined recursively on the structure of eq. $\texttt{prob}(eq, V) = True$, iff

- eq atomic event query and V occurs with negative polarity in eq;
- $eq = eq'$ $\texttt{in}$ $[b..e]$, or $eq = eq'$ $\texttt{before}$ e, and $\texttt{prob}(eq', V) = True$;
- $eq = eq'$ $\texttt{within}$ w and $\texttt{prob}(eq', V) = True$;
- $eq = \texttt{and}\{eq_1, \dots\ eq_n\}$ and $\exists i$, $1 \leq i \leq n$ such that $\texttt{prob}(eq_i, V) = True$;
- $eq = \texttt{andthen}[eq_1, \dots, eq_n]$ or $eq = \texttt{andthen}[[eq_1, \dots, eq_n]]$, and $\exists i$, $1 \leq i \leq n$ such that $\texttt{prob}(eq_i, V) = True$;
- $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \dots, eq_n]]$ and $\exists i$, $1 \leq i \leq n$ such that $\texttt{prob}(eq_i, V) = True$ or $\texttt{prob}(q_{ij}, V) = True$;
- $eq = \texttt{or}\{eq_1, \dots\ eq_n\}$ and $\forall i$, $1 \leq i \leq n$: $\texttt{prob}(eq_i, V) = True$;
- $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$ and $\texttt{prob}(eq_2, V) = True$;
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any var}\ X_1, \dots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ \{eq''\}$ and $\texttt{prob}(eq', V) = True$ or $\texttt{prob}(eq'', V) = True$;
- $eq = \texttt{times}\ (\texttt{atleast}|\texttt{atmost})?\ n\ \texttt{any var}\ X_1, \dots, \texttt{var}\ X_k\ \{eq'\}\ \texttt{during}\ [b..e]$ and $\texttt{prob}(eq', V) = True$;

- $eq =$ `every` n `any var` $X_1, \ldots,$ `var` X_k $\{eq'\}$ and $\texttt{prob}(eq', V) = True$;
- $eq =$ `withrank` n `any var` $X_1, \ldots,$ `var` X_k $\{eq'\}$ and $\texttt{prob}(eq', V) = True$;
- $eq =$ `last` $\{eq_1\}$ `during` $\{eq_2\}$ and $\texttt{prob}(eq_1, V) = True$ or $\texttt{prob}(eq_2, V) = True$;
- $eq =$ `last` $\{eq_1\}$ `during` $[b..e]$ and $\texttt{prob}(eq_1, V) = True$;
- $eq = (\texttt{atleast}|\texttt{atmost})?\, m$ `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $\{eq'\}$ and $\texttt{prob}(eq', V) = True$ or $\forall i$, $1 \leq i \leq n$: $\texttt{prob}(eq_i, V) = True$.

For all other cases, $\texttt{prob}(eq, V) = False$. Actually, predicate `prob` states whether a variable can be considered as having negative polarity in a (composite) event query.

Every XChange rule can be transformed into disjunctive normal form. Based on this result and using the predicate defined above, the range restriction of XChange rules can be formalised by the following definitions:

Definition 4.11 (Range Restriction of XChange Event-Raising Rules)
Let R be an XChange event-raising rule and $R' = t^e \leftarrow_r \bigvee_j q_j \leftarrow_r Eq$ its disjunctive normal form. R is range restricted iff

- $\forall V$ variable occurring in t^e with positive polarity, V occurs in Eq such that $\texttt{prob}(Eq, V) = True$ or in each of the q_j with negative polarity;
- $\forall V$ variable occurring in Eq or in at least one q_j with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq and q_j, V is also attributed as optional in all occurrences in Eq, q_j, and t^e.

Definition 4.12 (Range Restriction of XChange Transaction Rules)
Let R be an XChange transaction rule and $R' = \bigvee_k a_k \leftarrow_r \bigvee_j q_j \leftarrow_r Eq$ its disjunctive normal form. R is range restricted iff

- $\forall V$ variable occurring in one of the construct terms of a_k (that is, in one of the update or event terms of a_{kl}, where $a_k = \bigwedge_l a_{kl}$) with positive polarity, V occurs with negative polarity in Eq and/or in each of the q_j, and/or occurs with negative polarity in at least one of the a_{kl}, $1 \leq l \leq p$, where V in a_{kp};
- $\forall V$ variable occurring in Eq or in at least one q_j with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq and q_j, V is also attributed as optional in all occurrences in Eq, q_j, and a_k;
- $\forall V$ variable occurring in the subjacent query term of an update term a_{kl} with attribute optional and with negative polarity, and without another non-optional, with negative polarity occurrence of V in Eq, q_j, and a_{kw} with $1 \leq w \leq l-1$, V is also attributed as optional in all occurrences in the construct terms of a_{kl}.

An XChange program P is range restricted if all rules $R \in P$ are range restricted.

CHAPTER

FIVE

Semantics

The constructs offered by the reactive language XChange have been introduced in Chapter 4; grammar rules provide their syntax and examples explain their informal meaning. This chapter turns to defining the declarative semantics (Section 5.1) of the three parts of an XChange reactive rule: event query part (Section 5.1.1), Web query part (Section 5.1.2), and update part (Section 5.1.3). The chapter discusses also the evaluation of XChange event queries (Section 5.2.1), the underlying ideas of evaluating Web queries (Section 5.2.2), and of executing XChange updates (Section 5.2.3).

5.1 Declarative Semantics

The semantics of XChange event queries has been given in an informal manner in the previous chapter; this section defines their formal, declarative semantics, which is unambiguous and clear. Declarative semantics is beneficial to avoid misinterpretations of language constructs by both users and implementors. Also, declarative semantics provides a basis for formal proofs of language properties.

Notations are introduced for simplifying the definition of the declarative semantics. Recall that an XChange program is a set of rules, denoted $P = \{Rr_1, \ldots, Rr_m, Tr_1, \ldots, Tr_n, Dr_1, \ldots, Dr_p\}$, where

- Rr_i, $0 \le i \le m$, are event-raising rules of the form $t^e \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (t^e is an event term, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Tr_j, $0 \le j \le n$ are transaction rules of the form $tra \leftarrow_r \mathcal{Q} \leftarrow_r eq$ (tra is a transaction specification, $\mathcal{Q}$ an Xcerpt query, and eq an event query),
- Dr_k, $0 \le k \le p$ are Xcerpt rules of the form $t^c \leftarrow \mathcal{Q}$ (t^c is a construct term and $\mathcal{Q}$ an Xcerpt query), and
- $1 \le m + n$.

Some notes on XChange rules are needed here. An XChange program must contain at least one reactive rule (event-raising rule or transaction rule); this motivates $1 \le m + n$ for an XChange program P. Deductive rules are Xcerpt rules and not Xcerpt goals, that is rules Dr_k are views over persistent data that are materialised when they are needed. Deductive rules are not mandatory for an XChange program (that is why $0 \le k \le p$ for deductive rules Dr_k).

The set $\mathcal{T}$ denotes the set of all terms, $\mathcal{T}^u \subsetneq \mathcal{T}$ the set of all update terms, $\mathcal{T}^q \subsetneq \mathcal{T}$ the set of all query terms, and $\mathcal{T}^d \subsetneq \mathcal{T}$ the set of all data terms (*note* that in [125], $\mathcal{T}$ does not include update terms). The notation $\leftarrow_r$ has been chosen for the formal representation of reactive rules so as to differentiate between reactive and deductive (inference) rules in XChange.

The whole development of the language XChange follows a component-based approach, as it combines an event language, a query language, and an update language. The same approach is followed in defining the declarative semantics of XChange: Section 5.1.1 defines the declarative semantics of event queries, Section 5.1.2 gives an overview on the declarative semantics of Web queries (i.e. Xcerpt queries), and Section 5.1.3 discusses the declarative semantics of updates in XChange. The semantics of an XChange (reactive) rule follows immediately from the semantics of its components; the 'glue' between these rule components is given by the *substitution sets* for the variables occurring in these parts. Adaption or extension of one of the component languages (event query, Web query, and update languages) of XChange need to accommodate their semantics so as to keep the simple 'communication' inside a rule through substitution sets.

5.1.1 Semantics of Event Queries

The declarative semantics of XChange event query language is defined as a ternary relation between event queries, answers, and the stream of incoming event messages. This approach to the semantics is a 'special' model theoretical semantics; it is different from a model theoretic satisfaction relation usually encountered in logic programming languages because of the advanced constructs of the event query language. Following the later approach, answers to event queries would be obtained by applying substitutions to the event query. However, due to partial specifications in event queries this is not always the case. For example, by just applying a substitution (set) to an event query $eq = andthen[[eq_1, eq_2]]$ the atomic events that occur between the events answering eq_1 and eq_2, respectively, are not generated, although they are contained in the answer to eq. This is because answers to event queries contain more atomic events than actually specified through an event query's constituent atomic event queries.

A short discussion on time points and durations is given before formalising the answers to them, the incoming event stream, and the answering relation between them. The declarative semantics of event queries has been developed together with Prof. Dr. François Bry and Michael Eckert: Prof. Dr. François Bry has supervised the development of the whole XChange project. Michael Eckert has written his master's thesis under the double supervision of the author and Prof. Dr. François Bry. The declarative semantics of XChange event queries has been previously published in [76, 18]. However, the declarative semantics presented in the cited work does not cover all kinds of XChange event queries. The semantics of all event query language constructs is first defined in this thesis. The declarative semantics provide a sound basis for formal proofs of event language properties; this section ends with a formal proof of the fact that XChange legal event queries need only a bounded event stream for their evaluation (Section 5.1.1).

Time Points and Durations

A time domain $(\mathbb{T}, \mathbb{D})$ is used for interpreting time points and durations (lengths of time): Time points are interpreted as objects of $\mathbb{T}$. Durations are interpreted as objects of $\mathbb{D}$. For enhancing the readability, in defining the declarative semantics of XChange event queries time points and durations are considered as interpreted objects (instead of using the interpretation of the syntactical representation of time points and durations, respectively). A formally more correct approach would be to consider e.g. time points tp in the specification of event queries and $\mathcal{I}(tp)$ in the definition of event queries' semantics, where $\mathcal{I}$ is an interpretation function for time objects. For working with time points and durations, the following relations and conditions on the time domain $(\mathbb{T}, \mathbb{D})$ are needed for the declarative semantics presented in this section.

(i) An equality relation $=$ for both time points and durations.

(ii) A total order $<$ on time points; $t_1 < t_2$ expresses that time point t_1 is before time point t_2 on the time axis. A smallest time point exists, but no greatest time point.

(iii) The time difference $-$ between two time points; $t_2 - t_1$ (with $t_1 < t_2$) is a duration of time.

(iv) Durations can be compared with $<$; $w_1 < w_2$ expresses that w_1 is shorter than w_2.

(v) Addition $+$ of a duration to a time point results in a time point; $t_2 = t_1 + w$ (where t_1 time point and w duration) is a time point such that $t_2 - t_1 = w$.

(vi) A minimum min and a maximum max function for time points (from a set of time points return a time point that represents the minimum, maximum respectively, of them).

Event Stream

The *stream of incoming events*, or *event stream* for short, is the sequence of all atomic events an XChange-aware Web site receives. An atomic event a is a data term d that has been received at point $r \in \mathbb{T}$ (the occurrence time of the atomic event). The reception time point of events is written in superscript notation; thus, $a = d^r$. The domain of data terms $\mathcal{T}^d$ is defined in [125]. For the occurrence time r of an atomic event $a = d^r$, the notation $occ(a)$ is also used in this work for easing the readability.

Event queries are forward-looking (cf. Section 4.4.1), they do not have the ability to query events that have been received before their registration. Thus, for a given event query *eq*, all atomic events received after its registration form *eq*'s event stream. For readability reasons, $\mathcal{E}$ is used for denoting *eq*'s event stream, instead of $\mathcal{E}_{eq}$. From now on, each event stream used for defining the declarative semantics is considered relative to a given event query; the event query will be clear from the context.

An event stream $\mathcal{E}$ is a finite sequence $\langle a_1, a_2, \ldots a_n \rangle_b^e$ containing all atomic events $a_i = d_i^{ri}$ a reactive Web site is made aware of in the time interval $[b..e]$. The begin and end time points of the time interval are written in subscript notation and superscript notation, respectively. This extends the notation for atomic events where just a single time point, the occurrence time of the atomic event, is needed in representation.

The atomic events of an event stream have the following properties:
(i) $r_1 < \cdots < r_n$, i.e. they are totally ordered with respect to their occurrence time, and
(ii) $b \leq r_1$ and $r_n \leq e$, i.e. the atomic events of an event stream lie inside the interval $[b..e]$.
Note that (i) corresponds to the assumption that no two atomic events happen simultaneously.

Given an event query *eq*, for defining its correct semantics, the beginning time of its event stream is the time at which the event query has been registered to the system. For this work, $\mathcal{E}$ is not considered as the infinite sequence of atomic events a reactive system receives after registering *eq*. For evaluating an event query *eq* at some point in time t_{eval}, all events needed in the evaluation have a lower temporal bound, i.e. *eq*'s registration time, and an upper temporal bound, i.e. t_{eval}. Thus, it suffices to consider a finite sequence.

Answers

As explained in Section 4.4.5, the notion of answer to an event query is twofold, i.e. is made of the sequence of all atomic events that participated to answering the event query and a substitution set for all variables with at least one defining occurrence in the event query. Formally, an answer to an event query *eq* is a tuple (s, Σ), where s is the *event sequence* of atomic events that enabled the successful evaluation of *eq* and Σ the corresponding *substitution set*.

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, where $b \leq r_1 < r_2 < \cdots < r_n \leq e$. Cf. Section 4.2.3, answers to atomic event queries – atomic events – have an occurrence time point, answers to composite event queries – composite events – stretch over a time interval. The notion of event sequence accommodates both cases: For atomic events the beginning and ending time of the event sequence represent a single time point, the occurrence time of the atomic event (i.e. $b = e$). For composite events, the beginning time b of an event sequence represents the beginning time of the composite event, the ending time e represents the ending time of the composite event. *Note* that the time point e denotes the time at which the other parts of a reactive rule associated to the event query are evaluated. Recall that the beginning time b and ending time e of a composite event $c = \langle a_1, \ldots a_n \rangle_b^e$ are not necessarily r_1 and r_n, i.e. the reception times of the first and last, respectively, atomic events that are part of the event sequence representing the composite event (an explanatory example has been given in Section 4.2.3). In the following, *begin*(s) and *end*(s) are used for denoting the beginning time and ending time, respectively, of en event sequence s.

For defining the semantics of event queries, the notion of *event subsequences* and the *union of event sequences* are used; their formalisation follow.
Event Subsequences. A subsequence relation between event sequences is represented in this work with the (round) inclusion sign $\subsetneq$. Thus, $s \subsetneq s'$ denotes that s is a subsequence of s'. Formally, the subsequence relation is defined as follows:

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

(i) $\{a_1,\dots a_n\} \subsetneq \{a'_1,\dots a'_m\}$, and
(ii) $b' \leq b$ and $e \leq e'$.

The definition above states that the atomic events of a sequence s are to be found in the sequence s' if $s \subsetneq s'$. However, an event sequence s can also be a *complete subsequence* of a sequence s', i.e. s contains *all* atomic events from s' that lie between s's beginning and ending time. The complete subsequence relation between event sequences s and s' is written with a squared inclusion sign, $s \sqsubset s'$. The relation is defined as follows:

$\langle a_1,\dots a_n\rangle_b^e \sqsubset \langle a'_1,\dots a'_m\rangle_{b'}^{e'}$ if and only if
(i) $\{a_1,\dots a_n\} = \{a'_i \mid b \leq occ(a'_i) \leq e,\ 1 \leq i \leq m\}$, and
(ii) $b' \leq b$ and $e \leq e'$.

Note that $s \sqsubset s'$ implies $s \subsetneq s'$. Both relations $\subsetneq$ and $\sqsubset$ are reflexive and transitive.

Union of Event Sequences. Given two event sequences s and s', the union of the event sequences is denoted $s \cup s'$. The result $s'' = s \cup s'$ is an event sequence containing all events from s and s' and stretching over a time interval covering the intervals of s and s'. The union of event sequences is formalised for the n-ary (not just binary) case as follows:

$\langle a_1,\dots a_n\rangle_b^e \cup \langle a'_1,\dots a'_m\rangle_{b'}^{e'} =_{def} \langle a''_1,\dots a''_p\rangle_{b''}^{e''}$, where
(i) $\{a''_1,\dots a''_p\} = \{a_1,\dots a_n\} \cup \{a'_1,\dots a'_m\}$, and
(ii) $b'' \leq a''_1 < \dots < a''_p \leq e''$, and
(iii) $b'' = \min\{b,b'\}$ and $e'' = \max\{e,e'\}$.

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

Note that the event stream $\mathcal{E}$ is an event sequence; thus, the relations defined above do apply also for relating arbitrary sequences and the event stream.

Substitution Sets The notion of substitution sets introduced here is used also for the declarative semantics of condition and action parts of XChange reactive rules. It is important to use the same notion of substitution sets since the components of XChange reactive rules and the head and body of deductive rules communicate through substitution sets. To accommodate with the declarative semantics of the query language Xcerpt, which is integrated into XChange, the notion of substitution sets used in [125] is adopted for this work.

Substitution. A *substitution* maps variable names to construct terms. As the number of variables occurring in a term is always finite, the substitution here represents a finite mapping (function); in general, a substitution gives assignments for *all* variable names but its description covers just those of interest (e.g. variables occurring in a term, in our case). Substitutions are denoted in this work as variable assignments instead of as functions (e.g. $\{X \mapsto f\{\text{"}content\text{"}\}, Y \mapsto g\{\text{"}some\ content\text{"}\}\}$). Recall that the same denotation has been used in the previous sections for giving the informal meaning of XChange constructs through examples.

Substitutions are means for communicating event query and Web query result data to the action part of the reactive rules so as to *construct* event messages to be sent and/or to perform updates that depend on these substitutions. For constructing new data (to be sent or to be inserted) and to determine the subjacent query term of an update term, the obtained substitutions are *applied* to construct terms and query terms. *Applying* a substitution to a term means replacing all occurrences of variables for which an assignment is provided by the corresponding assignments; the result is a term ($\sigma(t)$ denotes the term obtained by applying σ to t). However, construct terms (specifying patterns for event messages, for views over persistent data, or for data that is to be inserted) may contain grouping constructs such as `all` or `some` (cf. Section 2.4.2 and Section 4.6), case in which *all* assignments for the variables in the construct term are needed. Thus, to accommodate such requirements, *substitution sets* are used in this work instead of substitutions.

Substitution Set. A *substitution set* Σ is a (finite) set of substitutions $\sigma_1,\dots,\sigma_n$ (denoted $\Sigma = \{\sigma_1,\dots,\sigma_n\}$). In the following, upper case greek letters (like Σ) denote substitution sets, while lower case greek letters (like σ) denote substitutions. Like substitutions, substitution sets can be *applied* to a term (query, construct, or update term) but the result is in general a set of terms (called *instances* of the respective query, construct, or update term). The definitions of substitution and substitution set application to query terms and construct terms are given in [125], Section 7.3.2 and Section 7.3.3. $\Sigma(t)$ denotes the result of applying the substitution set Σ to the term t.

Maximal Substitution Set. Moreover, the correct treatment of the grouping construct `all` requires also the substitution set to be *maximal*. An intuitive meaning of this property has been given in Section 4.4.5 when discussing the answers to XChange event queries. For an event query *eq*, a substitution set Σ answering *eq* is *maximal*, if there exists no substitution set Υ answering *eq* such that Σ is a proper subset of Υ. A formal definition of maximal substitution sets can be found in [125] (Section 7.3, Definition 7.1 on page 147).

Restriction of a Substitution Set. The *restriction* of a substitution set Σ to a set of variables V (denoted $\Sigma\mid_V$) is the set of substitutions of Σ for the variables of V, and undefined ($\perp$) for all other variables:

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

Defining Occurrence of Variables. The substitution set that is communicated between rule components and that is used for formalising the semantics of these components contains *only* substitutions for the variables having at least one *defining occurrence* in the (event or Web) query. Determining whether a variable occurring in the event part (or condition part) of a rule can be used in the condition and action part (or, just in the action part, respectively) of the rule or not, is realised by means of (positive or negative) polarities that are associated with each variable occurrence. Explanations of polarity of variables in XChange terms have been given in Section 4.4 and Section 4.8.4.

Answering Relation

The declarative semantics of event queries is defined as a ternary relation between event query *eq*, answer (s,Σ), and event stream $\mathcal{E}$. Given an event query *eq* and the stream $\mathcal{E}$ of incoming events received after its registration, the *answering relation* tells whether a tuple (s,Σ) is an answer to *eq* under $\mathcal{E}$. The answering relation is denoted $eq \lhd_{\mathcal{E}} (s,\Sigma)$ meaning '*eq* is answered by (s,Σ) under $\mathcal{E}$'. *Note* that the event stream $\mathcal{E}$ needs to be incorporated into the answering relation as the answers to *eq* depend on $\mathcal{E}$.

Consider *eq* an XChange event query. The answering relation $\lhd_{\mathcal{E}}$ is defined inductively on *eq*: The induction base is an XChange atomic query. The induction step uses case distinction on the top-level event composition construct or temporal restriction. In the following V denotes the set of all variables having at least one defining occurrence in *eq*.

Atomic Event Query Case *eq* atomic event query (that is, $eq\in\mathcal{T}^q$, where $\mathcal{T}^q$ the set of all possible Xcerpt query terms). A tuple (s,Σ) is an answer to *eq* under $\mathcal{E}$ if and only if s contains a single atomic event d^r of $\mathcal{E}$ and *eq* matches the data term d under all substitutions from Σ. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $\forall x\in V$, $\forall\sigma\in\Sigma$: $\sigma(x)$ is defined and respects the restrictions on x,
(ii) $\forall t\in\Sigma(eq)$: $t\preceq d$, and
(iii) $s=\langle d^r\rangle_r^r \sqsubset \mathcal{E}$.
Note that the answer to an atomic event query contains in this work a sequence of a single atomic event. Thus, answers to atomic event queries and answers to composite event queries are treated in an uniform manner. This simplify the use of the atomic event query case as the base of the inductive definition.

Conjunction Case $eq=$ `and`$\{eq_1,\ldots\ eq_n\}$. A tuple (s,Σ) answers *eq* under $\mathcal{E}$ if and only if s is the union of n event sequences from $\mathcal{E}$ that answer the component event queries eq_i. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff $\exists\ s_1,\ldots s_n$ event sequences such that
(i) $eq_i \lhd_{\mathcal{E}} (s_i,\Sigma)$, $\forall i$: $1\le i\le n$, and
(ii) $s=\bigcup_{1\le i\le n} s_i$.
Note that $begin(s)=\min_{i=\overline{1..n}} begin(s_i)$ and $end(s)=\max_{i=\overline{1..n}} end(s_i)$ expressing that the composite event answer to *eq* stretches over the time interval covering all component atomic events. Also, the requirement that $s_1,\ldots s_n \subsetneq \mathcal{E}$ (i.e. event sequences contain only atomic events from $\mathcal{E}$) is actually given by the induction base (this note applies also for the other cases of induction).

Disjunction Case $eq=$ `or`$\{eq_1,\ldots\ eq_n\}$. A tuple (s,Σ) answers *eq* under $\mathcal{E}$ if and only if (s,Σ) answers one of the component event queries eq_i. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $\exists i, 1\le i\le n$: $eq_i \lhd_{\mathcal{E}} (s,\Sigma)$.

Temporally Ordered Conjunction In order to simplify the definition of the case temporally ordered conjunction, the variable arity construct is defined by reducing it to the binary case, both for `andthen` with complete `[ ]` and incomplete `[[ ]]` specifications.

Case $eq = \texttt{andthen}[eq_1, eq_2]$. A tuple (s, Σ) answers eq under the event stream $\mathcal{E}$ if and only if s is the union of two event sequences from $\mathcal{E}$ that answer the component event queries eq_i, where all atomic events constituting the answer to eq_1 happen before all atomic events constituting the answer to eq_2. Formally, $eq \lhd_{\mathcal{E}} (s, \Sigma)$ holds iff $\exists\, s_1, s_2$ event sequences such that

(i) $eq_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for $i = 1, 2$,

(ii) $s = s_1 \cup s_2$, and

(iii) $end(s_1) < begin(s_2)$.

Note that $begin(s) = begin(s_1)$ and $end(s) = end(s_2)$ are comprised by the above given conditions.

Case $eq = \texttt{andthen}[eq_1, eq_2, eq_3, \ldots eq_n]$, $n > 2$. For defining the case of total temporally ordered conjunction with variable arity (or n-ary), the definition is reduced to the one given above by applying the following rewriting rule:

$\texttt{andthen}[eq_1, eq_2, eq_3, \ldots eq_n] \mapsto \texttt{andthen}[eq_1, \texttt{andthen}[eq_2, eq_3, \ldots, eq_n]]$.

Case $eq = \texttt{andthen}[[eq_1, eq_2]]$. A tuple (s, Σ) answers eq under the event stream $\mathcal{E}$ if and only if s is the union of three event sequences from $\mathcal{E}$: two answering eq_1 and eq_2, respectively, and one representing a continuous extract s' of $\mathcal{E}$ containing all atomic events received between the answers to eq_1 and eq_2. *Note* that s' is used for gathering all events received in-between and, thus, capturing the meaning of partial specification. Formally, $eq \lhd_{\mathcal{E}} (s, \Sigma)$ holds iff $\exists\, s_1, s', s_2$ event sequences such that

(i) $eq_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for $i = 1, 2$,

(ii) $s = s_1 \cup s' \cup s_2$,

(iii) $end(s_1) \leq begin(s_2)$,

(iv) $begin(s') = end(s_1)$ and $end(s') = begin(s_2)$, and

(v) $s' \sqsubset \mathcal{E}$.

Note that, like in the previous case, $begin(s) = begin(s_1)$ and $end(s) = end(s_2)$ are comprised by the above given conditions.

Case $eq = \texttt{andthen}[[eq_1, eq_2, eq_3, \ldots, eq_n]]$, $n > 2$. Like for the case of temporally ordered conjunction with total specification, the definition of n-ary partial temporally ordered conjunction is reduced to the one given above by applying the following rewriting rule:

$\texttt{andthen}[[eq_1, eq_2, eq_3, \ldots, eq_n]] \mapsto \texttt{andthen}[[eq_1, \texttt{andthen}[[eq_2, eq_3, \ldots, eq_n]]]]$.

Case $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q, eq_2]]$. A tuple (s, Σ) answers eq under the event stream $\mathcal{E}$ if and only if s is the union of three event sequences from $\mathcal{E}$: two answering eq_1 and eq_2, respectively, and one containing all atomic events received between the answers to eq_1 and eq_2 that answer q. Formally, $eq \lhd_{\mathcal{E}} (s, \Sigma)$ holds iff $\exists\, s_1, s', s_2$ event sequences such that

(i) $eq_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for $i = 1, 2$,

(ii) $s = s_1 \cup s' \cup s_2$,

(iii) $end(s_1) \leq begin(s_2)$,

(iv) $begin(s') = end(s_1)$ and $end(s') = begin(s_2)$,

(v) $\forall e$ in s': $q \lhd_{\mathcal{E}} (\langle e \rangle_{occ(e)}^{occ(e)}, \Sigma)$, and

(vi) if $\exists e'$ atomic event with $end(s_1) \leq occ(e') \leq begin(s_2)$ and $q \lhd_{\mathcal{E}} (\langle e' \rangle_{occ(e')}^{occ(e')}, \Sigma)$, then e' is in s'.

Case $eq = \texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]]$, $n > 2$. The definition of n-ary partial temporally ordered conjunction with collecting events having a given pattern is reduced to the one given above by applying the following rewriting rule:

$\texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]] \mapsto$
$\texttt{andthen}[[eq_1, \texttt{collect}\ q_{12}, \texttt{andthen}[[eq_2, \texttt{collect}\ q_{23}, eq_3, \ldots, eq_n]]]]$.

Absolute Temporal Restriction Absolute temporal restrictions of event queries have in XChange two forms: the monitoring time interval for answers to a given event query is specified either as an explicit time interval (with beginning and ending time) or specifying just the ending time (recall that in this case the beginning time is the event query's registration time). Thus, two cases need to be defined for absolute temporal restrictions.

Case $eq = eq'$ `in` $[b..e]$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' under $\mathcal{E}$ and the atomic events of s occur after time point b and before time point e. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq' \triangleleft_{\mathcal{E}} (s,\Sigma)$, and
(ii) $b \leq begin(s)$ and $end(s) \leq e$.

Case $eq = eq'$ `before` e. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' under $\mathcal{E}$ and the atomic events of s occur before time point e. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq' \triangleleft_{\mathcal{E}} (s,\Sigma)$, and
(ii) $begin(\mathcal{E}) \leq begin(s)$ and $end(s) \leq e$.

Relative Temporal Restriction Case $eq = eq'$ `within` w. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' under $\mathcal{E}$ and the length of time on which s stretches is less than or equal to the given duration w. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $q' \triangleleft_{\mathcal{E}} s,\Sigma$, and
(ii) $end(s) - begin(s) \leq w$.
Recall that time points and durations are already interpreted objects.

Variable Restriction Case $eq =$ `var` $X \rightarrow eq'$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' and all substitutions in Σ assign X to the composite event representation of s. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq' \triangleleft_{\mathcal{E}} (s,\Sigma)$,
(ii) $\forall \sigma \in \Sigma$: $\sigma(X) = xchange : event - seq[d_b, d_1, \ldots, d_n, d_e]$
where $s = \langle d_1^{r_1}, \ldots d_n^{r_n} \rangle_e^b$,
$d_b = xchange : beginning - time\{b\}$,
$d_e = xchange : ending - time\{e\}$.
Recall that a composite event answer to eq is represented as an XML document with root labelled $xchange : event - seq$ and containing the beginning and ending time of the event, and the atomic events from the event sequence s.

Exclusions For exclusion event queries, two forms are distinguished: the monitoring window over the event stream (a restriction of the event stream where answers to an event query are to be monitored) is given either by answers to a composite event query or by a (finite) time interval. Thus, two cases need to be defined for exclusion event queries.

Case $eq =$ `without` $\{eq_1\}$ `during` $\{eq_2\}$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq_2 and during the time interval determined by s no answer to eq_1 agrees with Σ on the variables with at least one defining occurrence in eq. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq_2 \triangleleft_{\mathcal{E}} (s,\Sigma)$,
(ii) $\forall s', s' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s')$ and $end(s') \leq end(s)$ and all Σ' with $eq_1 \triangleleft_{\mathcal{E}} (s',\Sigma')$ it holds that $\Sigma\, |_V \cap \Sigma'\, |_V = \emptyset$.

Case $eq =$ `without` $\{eq_1\}$ `during` $[b..e]$. This case is a slight modification of the previous one; as instead of the event query eq_2 a time interval is given, the first requirement from above is dropped out. Formally, $eq \triangleleft_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $\forall s', s' \subsetneq \mathcal{E}$ with $b \leq begin(s)$ and $e \leq end(s)$ and all Σ' with $eq_1 \triangleleft_{\mathcal{E}} (s',\Sigma')$ it holds that $\Sigma\, |_V \cap \Sigma'\, |_V = \emptyset$.
Note that the exclusion or negation of events in XChange's event query language represents the *classical negation* and not negation as failure. As the definitions above show, exclusion of events is rather easy to define; in contrary, negation as failure would be easy to implement but rather complicated to define.

Quantifications For a quantification event query $eq =$ `times` n `VarSpec` eq' `DuringSpec` to be successfully answered, eq' must be answered by at least n different answers (s_i, Σ_i). The substitution sets Σ_i must agree on all variables, except the existentially quantified variables (specified in `VarSpec` and following the keyword `any`).

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

$eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds if and only if there exist n event sequences $s_1,\dots s_n$ and substitution sets $\Sigma_1,\dots\Sigma_n$, and an event sequence s'' such that
(i) $s = s'' \cup \bigcup_{1\leq i\leq n} s_i$,
(ii) $eq'' \lhd_{\mathcal{E}} (s'',\Sigma)$,
(iii) $eq' \lhd_{\mathcal{E}} (s_i,\Sigma_i)$ $\forall i$, $1 \leq i \leq n$,
(iv) $begin(s'') \leq begin(s_i)$ and $end(s_i) \leq end(s'')$ $\forall i$, $1 \leq i \leq n$,
(v) $\Sigma_i \mid_{V\setminus\{X_1,\dots X_k\}} = \Sigma_j \mid_{V\setminus\{X_1,\dots X_k\}}$ $\forall i,j$, $1 \leq i < j \leq n$,
(vi) $\Sigma \subseteq \bigcup_{1\leq n} \Sigma_i$,
(vii) Σ_i is maximal (w.r.t. V and $eq' \lhd_{\mathcal{E}} (s_i,\Sigma_i)$) for all $1 \leq i \leq n$,
(viii) $s_i \neq s_j$ $\forall i,j$, $1 \leq i < j \leq n$,
(ix) if $\exists s'$: $s' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s')$ and $end(s') \leq end(s)$, and a Σ' with $\Sigma' \mid_{V\setminus\{X_1,\dots X_k\}} = \Sigma \mid_{V\setminus\{X_1,\dots X_k\}}$ such that $eq' \lhd_{\mathcal{E}} (s',\Sigma')$, then $s' = s_i$ and $\Sigma' \mid_V \subseteq \Sigma_i \mid_V$ for some $1 \leq i \leq n$.
Note that (ix) expresses the condition that there are no more than the n answers to eq' during the time interval determined by the answer to eq''.

Case $eq =$ `times atleast` n `any var` $X_1,\dots,$`var` X_k $\{eq'\}$ `during` $\{eq''\}$. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds if and only if there exist $m \geq n$ event sequences $s_1,\dots s_m$ and substitution sets $\Sigma_1,\dots\Sigma_m$, and an event sequence s'' such that the conditions from above hold with n replaced by m.

Case $eq =$ `times atmost` n `any var` $X_1,\dots,$`var` X_k $\{eq'\}$ `during` $\{eq''\}$. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds if and only if there exist $1 \leq m \leq n$ event sequences $s_1,\dots s_m$ and substitution sets $\Sigma_1,\dots\Sigma_m$, and an event sequence s'' such that the conditions from above hold with n replaced by m.

Cases $eq =$ `times(atleast|atmost)?` n `any var` $X_1,\dots,$`var` X_k $\{eq'\}$ `during` $[b..e]$. These cases are slight modifications of the cases discussed previously at quantification event queries. Their definitions are easily obtained when considering the empty sequence as having a duration, i.e. instead of $eq'' \lhd_{\mathcal{E}} (s'',\Sigma)$, the condition $s'' = \langle\rangle_b^e$ is used in the definitions above.

Repetitions Case $eq =$ `every` n `any var` $X_1,\dots,$`var` X_k $\{eq'\}$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' and there exist (s_i,Σ_i), $1 \leq i \leq n*m-1, 1 \leq m$ answers to eq' that agree with Σ on all variables, except the existential quantified variables. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq' \lhd_{\mathcal{E}} (s'',\Sigma)$,
(ii) $s = s''$ with $begin(s) = 0$,
(iii) $\exists 1 \leq m$, $\exists$ (s_i,Σ_i), $1 \leq i \leq n*m-1$ such that $\forall i, 1 \leq i \leq n*m-1$, and $max_{i=\overline{1,n*m-1}} end(s_i) \leq end(s)$, $\forall \Sigma_i$ maximal: $eq' \lhd_{\mathcal{E}} (s_i,\Sigma_i)$ with $\Sigma_i \mid_{V\setminus\{X_1,\dots X_k\}} = \Sigma \mid_{V\setminus\{X_1,\dots X_k\}}$, and
(iv) $\forall s'$ with $0 \leq end(s') \leq end(s)$ and all Σ' with $eq' \lhd_{\mathcal{E}} (s',\Sigma')$ it holds that $\Sigma \mid_V \cap \Sigma' \mid_V = \emptyset$.
Conditions (iii),(iv) state that (s,Σ) is the $n*m$ (i.e. multiple of n) answer to eq'.

Ranks In the event query language, one can specify ranks by means of a given position for events in the event stream or by specifying interest in the last event with a given pattern that is received in a monitoring time interval. Recall that ranks can be specified in XChange only for atomic events (cf. Section 4.4.3); that is, in the next cases eq' and eq_1 are *atomic* event queries. The notation $occ(s)$ is used for denoting the occurrence time of the atomic event represented as sequence s (recall that $begin(s) = end(s) = occ(s)$ in this case).

Case $eq =$ `withrank` n `any var` $X_1,\dots,$`var` X_k $\{eq'\}$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) answers eq' and there exist exactly $n-1$ tuples (s_i,Σ_i), $1 \leq i \leq n-1$ answers to eq' that agree with Σ on all variables except the existential quantified variables. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff
(i) $eq' \lhd_{\mathcal{E}} (s'',\Sigma)$,
(ii) $s = s''$ with $begin(s) = 0$ ($end(s) = occ(s'')$),
(iii) $\exists 1 \leq m$, $\exists$ (s_i,Σ_i), $1 \leq i \leq n-1$ such that $\forall i, 1 \leq i \leq n-1$, and $max_{i=\overline{1,n-1}} occ(s_i) \leq occ(s'')$, $\forall \Sigma_i$ maximal: $eq' \lhd_{\mathcal{E}} (s_i,\Sigma_i)$ with $\Sigma_i \mid_{V\setminus\{X_1,\dots X_k\}} = \Sigma \mid_{V\setminus\{X_1,\dots X_k\}}$, and
(iv) $\forall s'$ with $0 \leq occ(s') \leq occ(s'')$ and all Σ' with $eq' \lhd_{\mathcal{E}} (s',\Sigma')$ it holds that $\Sigma \mid_V \cap \Sigma' \mid_V = \emptyset$.

Case $eq =$ `last` $\{eq_1\}$ `during` $\{eq_2\}$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) is made of the answers to eq_1 and eq_2, where the answer to eq_1 is the last one during the answer to eq_2. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff $\exists s_1, s_2$ such that

(i) $s = s_1 \cup s_2$,

(ii) $eq_1 \lhd_{\mathcal{E}} (s_1,\Sigma)$,

(ii) $eq_2 \lhd_{\mathcal{E}} (s_2,\Sigma)$,

(iv) $begin(s_2) \leq occ(s_1) \leq end(s_2)$,

(v) $\forall s', s' \subsetneq \mathcal{E}$ with $occ(s_1) < occ(s') \leq end(s_2)$ and all Σ' with $eq_1 \lhd_{\mathcal{E}} (s',\Sigma')$ it holds that $\Sigma \cap \Sigma' = \emptyset$.

Condition (v) expresses that the tuple (s_1,Σ) is the last answer to eq_1 during the time interval determined by the answer to eq_2.

Case $eq =$ `last` $\{eq_1\}$ `during` $[b..e]$. The definition for this case can be easily obtained when replacing $eq_2 \lhd_{\mathcal{E}} (s_2,\Sigma)$ with $s_2 = \langle\rangle_b^e$ in the previously given definition.

Multiple Inclusions and Exclusions Case $eq = m$ `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $\{eq'\}$. A tuple (s,Σ) answers eq under the event stream $\mathcal{E}$ if and only if (s,Σ) represents the "union" of m tuples (s_i,Σ_i) answers to m of the given event queries where the substitutions agree on all variables with at least one defining occurrence except the variables following `any`, and the others $n-m$ event queries have no answer during the time interval determined by the answer to eq'. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds iff there exist m event sequences $s_1, \ldots s_m$ and substitution sets $\Sigma_1, \ldots \Sigma_m$ and a injective mapping $\iota : \{1, \ldots m\} \to \{1, \ldots n\}$, and an event sequence s' such that

(i) $s = s' \cup \bigcup_{1 \leq i \leq m} s_i$,

(ii) $eq' \lhd_{\mathcal{E}} (s',\Sigma)$,

(iii) $eq_{\iota(i)} \lhd_{\mathcal{E}} (s_i,\Sigma_i)\ \forall i,\ 1 \leq i \leq m$,

(iv) $begin(s) \leq begin(s_i)$ and $end(s_i) \leq end(s)\ \forall i,\ 1 \leq i \leq m$,

(v) $\Sigma_i |_{V \setminus \{X_1, \ldots X_k\}} = \Sigma_j |_{V \setminus \{X_1, \ldots X_k\}}\ \forall i,\ 1 \leq i < j \leq m$,

(vi) $\Sigma \subseteq \bigcup_{1 \leq m} \Sigma_i$,

(vii) Σ_i is maximal (w.r.t. V and $eq_i \lhd_{\mathcal{E}} (s_i,\Sigma_i)$) $\forall i,\ 1 \leq i \leq n$,

(viii) if $\exists s'', s'' \subsetneq \mathcal{E}$ with $begin(s) \leq begin(s'')$ and $end(s'') \leq end(s)$, and a Σ' with $\Sigma'' |_{V \setminus \{X_1, \ldots X_k\}} = \Sigma |_{V \setminus \{X_1, \ldots X_k\}}$ and a $1 \leq j \leq n$ such that $eq_j \lhd_{\mathcal{E}} (s'',\Sigma'')$, then $j = \iota(i)$ for some $1 \leq i \leq m$.

The other two forms of multiple inclusions and exclusions (i.e. `atleast`, `atmost`) where the monitoring time interval is given through a composite event query are defined in the same manner as above, but with slight modifications.

Case $eq =$ `atleast` m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $\{eq'\}$. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds if and only if there exist $p \geq m$, event sequences $s_1, \ldots s_p$, substitution sets $\Sigma_1, \ldots \Sigma_p$, and an injective mapping $\iota : \{1, \ldots p\} \to \{1, \ldots n\}$, and an event sequence s' such that the conditions from above hold with n replaced by p.

Case $eq =$ `atmost` m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $\{eq'\}$. Formally, $eq \lhd_{\mathcal{E}} (s,\Sigma)$ holds if and only if there exist $1 \leq p \leq m$, event sequences $s_1, \ldots s_p$ and substitution sets $\Sigma_1, \ldots \Sigma_p$, and an injective mapping $\iota : \{1, \ldots p\} \to \{1, \ldots n\}$, and an event sequence s' such that the conditions from above hold with n replaced by p.

A monitoring time interval can be explicitly specified for multiple inclusions and exclusions event queries; thus, other three cases (corresponding to exactly, `atleast`, and `atmost` m event queries out of n) need to be defined. Cases $eq =$ `(atleast|atmost)?` m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots, eq_n\}$ `during` $[b..e]$. Their definition can be easily obtained when replacing $eq' \lhd_{\mathcal{E}} (s',\Sigma)$ with $s' = \langle\rangle_b^e$ in the previously given definitions.

The declarative semantics provide a sound basis for formal proofs about event language properties. It has been used for proving that, in order to evaluate any legal event query eq at some time tp correctly, only events of bounded lifespan are necessary; that is, it suffices to consider the restriction $\mathcal{E} |_{tp-\beta}^{tp}$ of the event stream $\mathcal{E}$ to a time interval $[(tp-\beta)\ ..\ tp]$. The time bound β (a duration) is only determined from eq and does not depend on the stream of incoming events $\mathcal{E}$. The formal proof of the property is given in the next section.

Bounded Event Lifespan

This section is dedicated to prove that for finding answers to XChange legal event queries only events of a *bounded restriction of the event stream* are needed; it gives a formal proof of the promised legal event queries' property of keeping the clear cut between persistent and volatile data.

XChange legal event queries have been introduced in Section 4.4.4; recall that the following XChange event queries are legal:

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

XChange event queries do not have the ability to query events that have been received before their registration. Thus, for a given (legal) event query *eq*, all atomic events received after its registration form the *eq*'s event stream $\mathcal{E}$; that is, for proving that legal event queries need only events from bounded event stream for their evaluation, one can consider each event query individually (with its 'own' event stream).

Notation. An extract of the event stream $\mathcal{E} = \langle a_1, a_2, \ldots a_n \rangle_{tb}^{te}$ starting at time point b and ending at time point e is denoted as $\mathcal{E} \mid_b^e = \langle a_i, a_{i+1}, \ldots a_j \rangle_{b'}^{e'}$ where $b' \leq occ(a_i) \leq occ(a_{i+1}) \leq \cdots \leq occ(a_j) \leq e'$, $b' = \min\{tb, b\}$ and $e' = \max\{te, e\}$, and $\mathcal{E} \mid_b^e$ contains all events of $\mathcal{E}$ with occurrence time between b' and e'. Also, $\mathsf{Legal\_EvQ}$ and EvQ denote the set of possible, syntactically correct XChange legal event queries and event queries, respectively.

Theorem 5.1 (Bounded Event Lifespan)
For all legal event queries $eq \in \mathsf{Legal\_EvQ}$, there exists a time bound $\beta \in \mathbb{D}$ (a duration), such that for all time points $tp \in \mathbb{T}$, all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq tp$), and all answers (s, Σ) with $end(s) = tp$ holds that:

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

Proof. The idea of the proof is to divide the work into two: Prove that for some given legal event query *eq*, a bound β exists for the duration of its answer sequences. Then, prove that for finding an answer (s, Σ) to an event query *eq* only the events of $\mathcal{E}$ occurring between $begin(s)$ and $end(s)$ are needed; events occurring before $begin(s)$ or after $end(s)$ do not influence the fact that (s, Σ) answers *eq*.

These claims correspond to the two lemmas given next. The proof of this theorem follows from Lemma 5.2 and Lemma 5.4. □

Lemma 5.2 (Bound for Answers)
For all legal event queries $eq \in \mathsf{Legal\_EvQ}$, there exists a time bound $\beta \in \mathbb{D}$ (a duration), such that for all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq tp$), and all answers (s, Σ) with $end(s) = tp$ holds that:

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

Proof. For proving that the duration of all answer sequences of a legal event query *eq* has a time bound, a case distinction on the definition of $\mathsf{Legal\_EvQ}$ is made. *Note* that for an answer (s, Σ) to *eq*, $begin(s)$ and $end(s)$ are finite time points.

Case *eq* atomic event query, that is $eq \in \mathcal{T}^q$. Let $\beta = 0$. An answer to *eq* contains an event sequence of the form $s = \langle d^r \rangle_r^r$ (cf. definition of $\lhd_{\mathcal{E}}$ for *eq* atomic event query); thus, $end(s) - begin(s) = r - r = 0 \leq \beta$.

Case $eq = eq'$ `in` $[b..e]$. Let $\beta = e - b$. An answer (s, Σ) to *eq* satisfies the condition $b \leq begin(s)$ and $end(s) \leq e$ (cf. definition of $\lhd_{\mathcal{E}}$ for such an absolute temporal restriction); thus, $end(s) - begin(s) \leq e - b = \beta$.

Case $eq = eq'$ `before` e. Let $\beta = e - begin(\mathcal{E})$. Remember that $begin(\mathcal{E})$ is a finite time point. An answer (s, Σ) to *eq* satisfies the condition $begin(\mathcal{E}) \leq begin(s)$ and $end(s) \leq e$ (cf. definition of $\lhd_{\mathcal{E}}$ for such an absolute temporal restriction); thus, $end(s) - begin(s) \leq e - begin(\mathcal{E}) = \beta$.

Case $eq = eq'$ `within` w. Let $\beta = w$. An answer (s,Σ) to eq satisfies the condition $end(s) - begin(s) \leq w$ (cf. definition of $\lhd_{\mathcal{E}}$ for relative temporal restriction); thus, $end(s) - begin(s) \leq w = \beta$.

Case $eq =$ `without` $\{eq_1\}$ `during` $[b..e]$. Let $\beta = e - b$. An answer (s,Σ) to eq satisfies the condition $b \leq begin(s)$ and $end(s) \leq e$ (cf. definition of $\lhd_{\mathcal{E}}$ for such an exclusion); thus, $end(s) - begin(s) \leq e - b = \beta$.

Case $eq =$ `times (atleast|atmost)?` n `any var` $X_1, \ldots,$ `var` X_k $\{eq'\}$ `during` $[b..e]$. Let $\beta = e - b$. From the definition of $\lhd_{\mathcal{E}}$, an answer (s,Σ) to eq satisfies $s = s'' \cup \bigcup_{1 \leq i \leq m} s_i$, $begin(s'') \leq begin(s_i)$, $end(s_i) \leq end(s'')$, and $s'' = \langle\rangle_b^e$. Thus, using these conditions: $end(s) - begin(s) \leq end(s'') - begin(s'') = e - b = \beta$.

Case $eq =$ `last` $\{eq_1\}$ `during` $[b..e]$. Analogously to the `times`-case, where $i = 1$.

Case $eq =$ `(atleast|atmost)?` m `of any var` $X_1, \ldots$ `var` X_k $\{eq_1, \ldots eq_n\}$ `during` $[b..e]$. Analogously to the `times`-case. □

The Lemma 5.4 given next applies not only for legal event queries, but for all XChange event queries; it shows that for any XChange event query only events occurring during $[begin(s)..end(s)]$ influence the fact that a tuple (s,Σ) is answer to eq or not. For easing the proof, the following proposition is formulated and proved so as to use its results in proving the lemma; the proposition states some properties of the subsequence relations $\sqsubset$ and $\subsetneq$ that have been defined in a previous section.

Proposition 5.3 (Properties of Subsequence Relations)
The following hold for the subsequence relations $\sqsubset$ and $\subsetneq$:

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

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

Proof. The proof is restricted to proving that the above given property holds for $\sqsubset$; the proof for $\subsetneq$ is analogous. Recall the definition of complete subsequence relation between event sequences s and s' ($s \sqsubset s'$):
$\langle a_1, \ldots a_n\rangle_b^e \sqsubset \langle a'_1, \ldots a'_m\rangle_{b'}^{e'}$ if and only if
(i) $\{a_1, \ldots a_n\} = \{a'_i \mid b \leq occ(a'_i) \leq e,\ 1 \leq i \leq m\}$, and
(ii) $b' \leq b$ and $e \leq e'$.

"$\Longrightarrow$": Given $s \sqsubset \mathcal{E}\,|_{begin(s)}^{end(s)}$. To show $s \sqsubset \mathcal{E}\,|_b^e$.

From the definition of restriction of $\mathcal{E}$ to a time interval, one obtains $\mathcal{E}\,|_{begin(s)}^{end(s)} \sqsubset \mathcal{E}\,|_b^e$. Now, based on the transitivity of $\sqsubset$ and $s \sqsubset \mathcal{E}\,|_{begin(s)}^{end(s)}$, one obtains $s \sqsubset \mathcal{E}\,|_b^e$.

"$\Longleftarrow$": Given $s \sqsubset \mathcal{E}\,|_b^e$. To show $s \sqsubset \mathcal{E}\,|_{begin(s)}^{end(s)}$.

Showing $s \sqsubset \mathcal{E}\,|_{begin(s)}^{end(s)}$ means showing that $\forall a$ atomic event in s, with $s \sqsubset \mathcal{E}\,|_b^e$, $\Rightarrow a$ in $\mathcal{E}\,|_{begin(s)}^{end(s)}$.

Let a an (arbitrary) atomic event in s. From $s \sqsubset \mathcal{E}\,|_b^e$, one obtains a in $\mathcal{E}\,|_b^e$. The following condition holds (because a in s): $begin(s) \leq occ(a) \leq end(s)$; now, based on the definition of restriction of $\mathcal{E}$ to a time interval, one obtains a in $\mathcal{E}\,|_{begin(s)}^{end(s)}$.

□

Lemma 5.4 (Bounded Extract of Event Stream)
For all event queries $eq \in \mathsf{EvQ}$, all event streams $\mathcal{E}$, and all answers (s,Σ) holds that:

$$eq \lhd_{\mathcal{E}} (s,\Sigma) \iff eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma).$$

Proof. Generalising $\mathcal{E}$ to $\mathcal{E}\,|_b^e$, the following are to be proven:
"$\Longrightarrow$": $\forall b \leq begin(s), \forall e \geq end(s). \quad eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma) \implies eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$, and
"$\Longleftarrow$": $\forall b \leq begin(s), \forall e \geq end(s). \quad eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma) \implies eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$.

The proof is by induction on eq; that is, as in the definition of the declarative semantics, for all (event query) cases the two above given statements need to be proven. Thus, the whole proof of the lemma would stretch over a couple of pages. Just a few cases are given here that give flavour of a pattern for giving the proof for the other cases.

Let $b \leq begin(s)$ and $e \geq end(s)$ two arbitrary time points.

<u>Case</u> eq atomic event query, that is $eq \in \mathcal{T}^q$.

"$\Longrightarrow$": Given $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$. To show $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$.

From $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$ and $eq \in \mathcal{T}^q$, the next given conditions follow: $begin(s) = end(s) = r$ and $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E} \mid_b^e$. Now, based on Proposition 5.3, one obtains $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E} \mid_r^r$.

"$\Longleftarrow$": Given $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$. To show $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$.

From $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$ and $eq \in \mathcal{T}^q$, the next given conditions follow: $begin(s) = end(s) = r$ and $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E} \mid_r^r$. Now, based on Proposition 5.3, one obtains $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E} \mid_b^e$.

<u>Case</u> $eq = \texttt{and}\{eq_1, \dots\ eq_n\}$.

"$\Longrightarrow$": Given $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$. To show $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$.

From $eq = \texttt{and}\{eq_1, \dots\ eq_n\}$ and $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$, the next given conditions follow: $eq_i \lhd_{\mathcal{E}|_b^e} (s_i,\Sigma)$, $\forall i :$ $1 \leq i \leq n$, and $s = \bigcup_{1\leq i \leq n} s_i$. Thus, one needs to prove that $\forall i, s_i \subsetneq \mathcal{E} \mid_{begin(s)}^{end(s)}$.

From $\forall i, s_i \subsetneq \mathcal{E} \mid_b^e$ and Proposition 5.3, one obtains $\forall i, s_i \subsetneq \mathcal{E} \mid_{begin(s_i)}^{end(s_i)}$. From $begin(s) = \min_{i=\overline{1..n}} begin(s_i)$ and $end(s) = \max_{i=\overline{1..n}} end(s_i)$, one obtains $\forall i, \mathcal{E} \mid_{begin(s_i)}^{end(s_i)} \subsetneq \mathcal{E} \mid_{begin(s)}^{end(s)}$; now, based on the transitivity of $\subsetneq$: $\forall i, s_i \subsetneq \mathcal{E} \mid_{begin(s)}^{end(s)}$.

"$\Longleftarrow$": Given $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$. To show $eq \lhd_{\mathcal{E}|_b^e} (s,\Sigma)$.

From $eq = \texttt{and}\{eq_1, \dots\ eq_n\}$ and $eq \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s,\Sigma)$, the next given conditions follow: $eq_i \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s_i,\Sigma)$, $\forall i :\quad 1 \leq i \leq n$, and $s = \bigcup_{1\leq i \leq n} s_i$. Thus, one needs to prove that $\forall i, s_i \subsetneq \mathcal{E} \mid_b^e$.

From $\forall i, s_i \subsetneq \mathcal{E} \mid_{begin(s)}^{end(s)}$ and Proposition 5.3, one obtains $\forall i, s_i \subsetneq \mathcal{E} \mid_b^e$.

<u>Cases</u> $eq = \texttt{andthen}[[eq_1, eq_2]]$ and $eq = \texttt{without}\ \{eq_1\}\ \texttt{during}\ \{eq_2\}$ are given in [76], Section 6.4, pages 83 - 84.

□

Lemma 5.4 has been used to prove the Theorem 5.1, which states that XChange legal event queries require bounded event stream for their evaluation. However, Lemma 5.4, by regarding not only legal event queries but (legal and 'illegal') event queries, has proved a useful property of the event query language of XChange: for determining an answer to an XChange event query, one needs not consider events that do not lie between the beginning and ending times of the answer (of its event sequence). For any changes or extensions to the constructs of XChange event queries, Lemma 5.4 needs to hold.

5.1.2 Semantics of Web Queries: Underlying Ideas

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

A model theory for the query language Xcerpt has been developed (see [125], Chapter 7), which follows the approach of classical Tarski-style semantics for first order logic. However, the distinctive features (such as the grouping constructs in the head of the rules and partial specifications of queries) of the language Xcerpt entailed considerable differences from the classical logic. Classical logic differentiate between *terms* (representing objects) and *atomic formulas* (representing statements about objects); though, Xcerpt terms are atomic formulas expressing the statement that the respective term exists. Informally, an *interpretation* is a set of data terms that specifies what data terms exist and a *model* is an interpretation containing the terms inferred by the given Xcerpt rules.

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

Example 5.1 (Formula Representation of Xcerpt Rules)
Consider the following set of Xcerpt rules:

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

The above given set of rules is represented as formula as follows:

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

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

Satisfaction and Models An atomic formula F is considered to be satisfied in interpretation M if and only if its ground instance (obtained by applying the substitutions of Σ to F) simulates into a term of I. The satisfaction of a term formula (i.e. atomic or compound formula) is defined recursively over its structure. The following definition is taken from [125], Section 7.4.2, on pages 151-152:

Definition 5.5 (Satisfaction, Model)

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

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

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

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

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

Given an Xcerpt program Pr, a model for Pr is an interpretation (I,Σ) where I contains *all* data terms that are inferred (or produced) by the rules of Pr. *Note* that I may contain also data terms that are unrelated to Pr. A concrete example for satisfaction of Xcerpt programs is given in [125], Chapter 7, pages 152-153. However, the focus in this section is on the parts of XChange programs that are expressed in Xcerpt. Recall that an XChange program has the form $P = \{Rr_1,\ldots,Rr_m,Tr_1,\ldots,Tr_n,Dr_1,\ldots,Dr_p\}$, where Rr_i, $1 \leq i \leq m$ and Trj, $1 \leq j \leq n$ are reactive rules that may have 'condition parts' specifying Xcerpt queries. Also, Dr_k, $1 \leq k \leq p$ are Xcerpt rules. Consider $\mathcal{Q}_l$, $1 \leq l \leq m+n$ the Xcerpt queries associated with the reactive rules of P. The interest is on the satisfaction of the formulas of the form:

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

where Dr_k is of the form $\quad t_k^c \leftarrow Q_k^r \quad$, for $1 \leq k \leq p$ and $d_h \in \mathcal{T}^d$ are data terms that represent the internalised Web resources that are specified in the queries. *Note* that the interest for XChange programs is not on the conjunction $\bigwedge_{1 \leq l \leq m+n}$ of the formulas given above. As the definition 5.5 covers also formulas of the form given above, the semantics of the 'condition parts' of XChange reactive rules is given by the model theoretical approach of Xcerpt.

Example 5.2 (Formula Representation of XChange Condition Parts and Deductive Rules)
Consider an XChange program P that consists of the following rules:

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

Note that P contains one transaction rule whose event query and transaction specification are not specified as they do not contribute to the formula representation of Web queries and deductive rules. The deductive rules of P are the Xcerpt rules of Example 5.1 with a slight modification of the data term. The above given 'condition part' and set of deductive rules is represented as formula as follows:

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

A fixpoint semantics for Xcerpt programs without negation is proposed in [125], Chapter 7, Section 5. A fixpoint operator T_P is defined by applications of which a fixpoint for Xcerpt programs is iteratively constructed. It is proved that the fixpoint of a program is also a model of the program (Theorem 7.10 on page 155).

5.1.3 Semantics of Updates

This section discusses the declarative semantics of XChange's update language; it presents the semantics of elementary and complex updates in XChange .

Recall that transactions – combining updates and events to be raised – can be specified in XChange. However, this work on declarative semantics is restricted to XChange updates, as the accent in this thesis is not on a language for distributed transactions on the Web; the thesis recognises the need for transactions through developed application scenarios and the components a transaction on the Web might have, and proposes a syntax for event-driven transactions. The declarative and operational semantics of transactions on the Web are outside the scope of this thesis.

The idea of the semantics for the update language is based on an interesting observation, namely that the XChange updates are an *elegant* and *easy* way for specifying data modifications as an alternative to an intentional specification, i.e. constructing the data after the update by means of deductive rules.

Semantics of Elementary Updates An XChange elementary update specification consists of a resource specification and an update term: The resource specification gives the location and names of the documents to be updated. The update term is a pattern for the data to be updated augmented with the desired update operations. The effect of an elementary update is that the data at the given resources has been 'refreshed' according to the given update term. The same result can be obtained when *constructing* the data after the update. In the context of XChange this means that an elementary update has the same effect as an Xcerpt *goal* that constructs the data after the update.

Given an elementary update u for modifying resources Res_i, $1 \leq i \leq n$ (a finite number of Web resources) with update term t^u, a corresponding Xcerpt goal G_u exists that constructs new data at Res_i. This (new) data constructed at Res_i is the data that would be obtained by applying t^u on the (old) data at Res_i. *Note* that data constructed by G_u overwrites the old data found at the resources. A set of rewriting rules have been recognised that rewrite an elementary update u into an Xcerpt goal G_u such that the effect of G_u is the same as the effect of u. The underlying ideas of the rewriting are discussed in detail in Section 5.2.3.

Thus, the semantics of an elementary update u can be reduced to the semantics of a corresponding Xcerpt goal G_u of the form $t^c_u \leftarrow \mathcal{Q}$. That is, the model theoretical semantics of Xcerpt, whose underlying ideas have been presented in the previous section, can be used for defining the semantics of elementary updates in XChange.

An elementary update u is transformed into a goal $t^c_u \leftarrow \mathcal{Q}_u$ and gets, thus, a formula representation $\forall^* \ll t^c_u \leftarrow \mathcal{Q}_u \gg \wedge \bigwedge_{1 \leq i \leq n} d_{Res_i}$, where d_{Res_i} are the data terms to be modified. The satisfaction of such a formula in an interpretation is defined in Definition 5.5. Intuitively, the model for the formula expressing the update contains the data terms after the update has been performed.

Example 5.3 (Declarative Semantics of Elementary Updates)
Consider the following update term:

```
bib {{
      book {{ price { var P replaceby var P * 1.5 } }}
}}
```

Assume that the data term to be updated contains books listed with their titles and prices; each book price needs to be modified. A goal corresponding to the above given update term has the following formula representation:

```
∀ P ≪bib { book { price { var P*1.5 }, all
    var O }, all var C } ← bib {{ book {{ price {
    var P }, var O }}, var C }} ≫ ∧
bib { currency {"Euro"}, book { title {"Linux in a Nutshell"}, price {"36"} },
book { title {"Data on the Web"}, price {"40"} } }
```

A model for the above given formula is $M = (I, \Sigma)$ where:

```
I = {
   bib { currency {"Euro"}, book { title {"Linux in a Nutshell"}, price {"54"} },
                            book { title {"Data on the Web"}, price {"60"} } },
   bib { currency {"Euro"}, book { title {"Linux in a Nutshell"}, price {"36"} },
                            book { title {"Data on the Web"}, price {"40"} } }
}
Σ = { {∅} }
```

The interpretation I contains the data terms before and after the update. The formula corresponding to the update term together with the data term to be modified represents an Xcerpt program that produces the data term where the book prices are replaced by new, bigger prices.

Semantics of Complex Updates Complex updates specify conjunctions or disjunctions of (elementary or complex) updates. Based on the formula representation of elementary updates and using the connectors $\wedge$ and $\vee$ the formula representation of complex updates are constructed. A formula of the form

(i) $F_1 \wedge \cdots \wedge F_n$ represents a complex update specifying conjunction of updates,

(ii) $F_1 \vee \cdots \vee F_n$ represents a complex update specifying disjunctions of updates,

where F_i is the formula representation of an elementary or complex update. As for elementary updates, the Definition 5.5 is used for determining if such a formula is satisfied in an interpretation or not. This approach is suitable for defining the semantics of (unordered) complex updates; the definition of the scope of variables in XChange update terms does not preclude it (cf. Section 4.7.2). Recall that for an unordered conjunction or disjunction of updates u_i, $2 \leq i \leq n$, the scope of variables used in update u_i is restricted to u_i, i.e. the bindings for the variables resulted from evaluating u_i can not be used in the evaluation of u_j with $i \neq j$.

Remark Observing that the effect of an elementary update can be 'simulated' by an Xcerpt goal, the model theoretical semantics of the query language Xcerpt has been used for defining the declarative semantics of the update language of XChange. However, this approach does not cover ordered conjunctions and disjunctions of (elementary or complex) updates. Ordered complex updates enforce an order for performing the given updates. Moreover, some specified updates might depend on updates that are to be executed before them (so as to use bindings for the variables obtained after update execution). These features can not be defined by means of a model theoretical semantics.

5.2 Operational Semantics

XChange programs are located at Web sites distributed over the network. By means of specified XChange rules, persistent data is updated and events are raised as reactions to (local or remote) events that have occurred. An XChange program at a Web site does not update only local Web resources and does not raise events only for its own 'usage'. Instead, updates to remote Web resources are requested to other XChange programs and events are raised and sent to one or more (remote) XChange programs; in turn, these events trigger updates at and raising of events to other XChange programs at Web sites. Thus, through local XChange programs, global behaviour is achieved.

Locally, executing XChange program(s) determines local state changes (at the Web site where the program runs); it can also request and/or trigger state changes at remote Web sites. Globally, executing XChange programs determine 'global' state changes that depend on the local ones (and the time point of execution). XChange programs specify state transitions on the Web. Thus, the operational semantics of XChange programs can be formalised by using approaches such as Petri nets or modal logics [70]. Also, work on logics for database transactions [37] might prove useful. At moment, XChange provides semantics for XChange rule execution and not for the global behaviour of XChange programs running at different Web sites; the latter is one of the perspectives for future work. Also, means for realising transaction management on the Web will determine investigating different coupling modes for XChange rules; possible coupling modes found in the literature have been presented in Section 3.1.2.

XChange assumes no priorities for rules; given an XChange program $P = \{Rr_1, \ldots, Rr_m, Tr_1, \ldots, Tr_n, Dr_1, \ldots, Dr_p\}$, where Rr_i, $0 \leq i \leq m$ and Tr_j, $0 \leq j \leq n$, with $1 \leq m+n$ are event-raising and transaction rules, and Dr_k, $0 \leq k \leq p$ are Xcerpt rules (i.e. deductive rules), upon reception of an event e each of the event queries of Rr_i and Tr_j are evaluated. If answers to some of the event queries are found, no order is given for evaluating the Web queries and executing the actions of the rules associated with these event queries.

The next sections discuss evaluation/execution of the three components of an XChange reactive rule; evaluation of deductive rules is touched on when discussing evaluation of Web queries. The evaluation of XChange event queries is discussed in Section 5.2.1; this work has been developed together with Prof. Dr. François Bry and Michael Eckert: Prof. Dr. François Bry has supervised the development of the whole XChange project. Michael Eckert has written his master's thesis under the double supervision of the author and Prof. Dr. François Bry. The evaluation of event queries has been previously published in

[76, 18]. Section 5.2.2 offers a very short discussion on evaluation of Web queries (and deductive rules). Section 5.2.3 ends this section with a discussion on executing XChange updates.

5.2.1 Evaluation of Event Queries

XChange event queries have been introduced in Section 4.4, where their informal semantics has been given through simple examples; Section 5.1.1 has defined the declarative semantics of XChange event queries. This section presents the approach taken in XChange for *evaluating event queries*. Given an XChange program $P = \{Rr_1, \ldots, Rr_m, Tr_1, \ldots, Tr_n, Dr_1, \ldots, Dr_p\}$, the event queries of P's reactive rules need to be evaluated before any Web query is evaluated or action is performed. Assume that $1 \leq s \leq m+n$ is the number of event query registered in the system through the program P; *note* that s is not always equal to $m+n$ as e.g. transaction rules may exist for updating data or raising events not necessarily as reactions to events. This part of the thesis discusses how event queries eq_k, $1 \leq k \leq s$ are evaluated in XChange; some eq_k are atomic event queries and some are composite event queries. After a general discussion on event query evaluation in XChange, the approach taken in XChange for evaluating atomic event queries and composite event queries is presented.

Generalities

Local Processing of Events and Evaluation of Event Queries XChange assumes no central processing of event queries as such an approach is not suitable on the Web. Instead, event queries are processed locally at each XChange-aware Web site. Each such Web site has its own local *event manager* for processing incoming events and evaluating event queries against the incoming event stream (volatile data).

Incremental Evaluation Event queries need to be evaluated in an *incremental* manner, as data (events) that are queried are received in a stream-like manner and are not persistent. Upon reception of events (represented as event messages) partial evaluations of event queries are computed; these partial 'results' are used in future evaluation steps done upon reception of other, future incoming events. In other words, evaluation work is used for future evaluation.

Logical Variables Variables are place holders for the data, in the fashion of logic programming variables are. They require equality when occurring more than ones in an (atomic or composite) event query. The evaluation of an XChange event query needs to yield the answers to the event query based on the received events; that includes finding maximal substitution sets for the variables in the event query. For this task, variables in event queries need to be properly treated. To the best of author's knowledge, related work on composite event query evaluation does not consider treatment of logical variables; new algorithms for event query evaluation are needed, as existing one can not be used in this setting or at least need considerable modifications.

Bounded Event Lifespan An essential aspect for event processing is that each reactive Web site controls its own event memory usage. In particular, which events and for how long they are kept in memory depends only on the event queries posed at a Web site. Event lifespans are automatically detected from the event queries already registered at a Web site. XChange legal event queries have been designed in such a way that no data on any event can be kept forever in memory, i.e. the event lifespan is *bounded*. Thus, based on the proof given in Section 5.1.1, the evaluation of event queries has also the task of discarding events when their lifespan expires.

Evaluation of Atomic Event Queries

As explained in Section 4.4, XChange atomic event queries are patterns for incoming event messages that are of interest for a Web site. The event manager of an XChange-aware Web site tries to *match* each incoming event message received with the currently posed atomic event queries (which themselves may be part of composite event queries). The matching of an atomic event query with an incoming event is based on the *Simulation Unification* [125], a novel unification method whose underlying ideas have been presented in Section 2.4.2. *Note* that the same method is used for *simulating* Web queries into persistent data terms and for *simulating* atomic event queries into volatile data terms (i.e. incoming event messages).

The outcome of evaluating an atomic event query *eq* against an incoming event message *msg* (i.e. the result of $eq \preceq msg$) is either the boolean value *False* (expressing unsuccessful evaluation) or a substitution set Σ for the variables with at least one defining occurrence in *eq* (expressing successful evaluation); in the latter case, the answer to *eq* is (msg, Σ).

Evaluation of Composite Event Queries

Evaluation of XChange composite event queries is done *locally* (at each XChange-aware Web site), in an *incremental manner* (partial composite event query evaluations are amended upon reception of new events), with *proper treatment of logical variables*, and *assuring a bounded event lifespan* (stored events are dispensed when their lifespan expires).

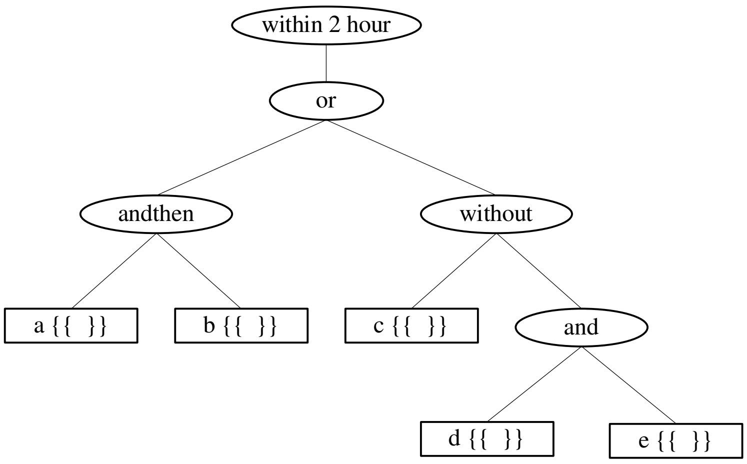
The issue of evaluating composite event queries, called composite event detection in the literature, has received considerable attention in the field of active database systems. Section 3.1.2 has presented the most popular approaches to (algorithms for) composite event detection found in the literature: *Petri nets*, *finite state automata*, and *tree with bottom-up flow of events*. For evaluating XChange composite event queries, a tree-based approach with bottom-up flow of event data is used. The reasons for choosing such an approach for XChange event queries include: The algorithms are easy to comprehend and, thus, to adapt to possible future changes or extensions of the XChange event query language. The evaluation is reasonably efficient and leaves room for an easier optimisation (such as event query rewriting) than the other two approaches. For a detailed comparison of the approach taken in XChange and related work on composite event detection, see [76], Sections 7.3.1 and 7.3.2.

Tree Representation For each composite event query eq_k, $1 \leq k \leq s$ an operator tree is constructed (by parsing and compiling eq_k): The leaves of the operator tree implement atomic event queries that are constituents of eq_k. The inner nodes implement XChange event language constructs such as `and`, `andthen`, `or`, `without`, or `before Time Point`. Examples of tree representation for XChange event queries follow. The compact notation introduced in Section 4.4.3 is used throughout this section.

Example 5.4 (Tree Representation of XChange Event Queries (1))
Consider the following composite event query:

```
or {
   andthen [ a{{}}, b{{}} ],
   without {
            c{{}}
           } during {
                 and { d{{}}, e{{}} }
                 }
} within 2 hour
```

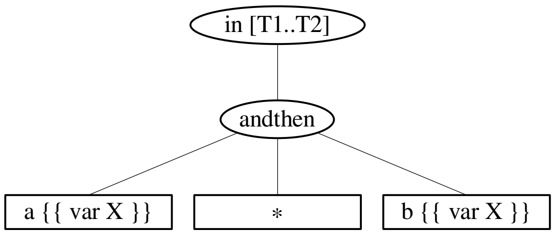
The operator tree of the above given composite event query is given next. The `andthen` event query has two constituent event queries and, thus, two child nodes (leaves because they are atomic). The same situation is for representing the conjunction event query, introduced by `and`. *Note* that the representation of `without` has two child nodes: one corresponding to the event query whose non-occurrence is to be detected, and one to the composite event query restricting the monitoring time interval. The same approach is used for the other composite event queries with a `during CompositeEventQuery` specification (e.g. for `times`).



Example 5.5 (Tree Representation of XChange Event Queries (2))
Consider now the following XChange composite event query specifying temporally ordered conjunction of events (where $T1$ and $T2$ represent time points):

```
andthen [[
      a {{ var X }},
      b {{ var X }}
]] in [T1..T2]
```

The operator tree of the `andthen` event query is given next. *Note* that the operator node for `andthen` has besides the two children corresponding to the given atomic event queries, a (middle) child node $*$ that matches any incoming atomic event (actually, $*$ implements an atomic event `xchange:event` $\{\{\}\}$); this is used for gathering all atomic events occurring between the answers to the two given atomic event queries.



Operator trees are used for detecting composite events that are answers to XChange event queries *eq*. Cf. Section 4.4.5 and Section 5.1.1, an answer to an XChange event query *eq* is a tuple consisting of a sequence of atomic events and a substitution set for the variables with at least one defining occurrence in *eq*. For evaluating *eq* in an incremental manner, partial evaluations of *eq* in form of event sequences and substitution sets are stored at some of the inner nodes of the operator tree. Such a storage at an inner node consists of all composite events from its child nodes that might be needed in future evaluation steps (upon reception of new events) to build an answer to the whole event query *eq*. Whether an inner node has such storage for event data or not depends on the language construct the node represents. For example,

- an operator tree representing binary `andthen` needs to store (e.g. as a list ordered by reception times of) events from its left child. Upon detection of an event answering from its right child, this event is

```
**SetOfCompositeEvents** evaluate( **AndNode** $n$, **AtomicEvent** a ) {
    // receive events from child nodes
    **SetOfCompositeEvents** newL := evaluate( n.leftChild, a );
    **SetOfCompositeEvents** newR := evaluate( n.rightChild, a );

    // compose composite events
    **SetOfCompositeEvents** answers := $\emptyset$;
    **foreach** $((s_L,\Sigma_L),(s_R,\Sigma_R))$ $\in$(newL $\times$ n.storageR) $\cup$
                                   (n.storageL $\times$ newR) $\cup$
                                   (newL $\times$ newR) {
        **SubstitutionSet** $\Sigma$ := $\Sigma_L \bowtie \Sigma_R$;
        **if** ($\Sigma \neq \emptyset$) answers := answers $\cup$ **new CompositeEvent**( $s_L \cup s_R$, $\Sigma$ );
    }

    // update event storage
    n.storageL := n.storageL $\cup$ newL;
    n.storageR := n.storageR $\cup$ newR;

    // forward composed events to parent node
    **return** answers;
}
```

Figure 5.1: Implementation of a (binary) `and` inner node in pseudo-code

paired up with all events in the storage of the `andthen` operator node. The outcome, i.e. composite events, is forwarded (or pushed to the level above) to the parent of the `andthen` operator node. Section 5.1.1 shown how variable arity `andthen` can be reduced to binary ones. (*Note* that it would suffice to store references to events, and not their copies.)

- an operator tree representing `or` does not store data on any event; whenever one of its child nodes detects an event, the event is forwarded to the parent node of the `or` operator node.

Composite events that reach the root of the operator tree of a composite event query *eq* represent answers to the whole event query *eq*. These composite events trigger the rule having as 'event part' the composite event query *eq*. Let's take a closer look at the evaluation that is done for composing atomic events injected at the leaves into composite events reaching the root. The issue of deleting events from inner nodes' storage is discussed afterwards.

Bottom-Up Event Data Flow Incoming atomic events are matched against each leaf node of the operator tree representing an event query to evaluate. The answers to the leaf nodes' event queries are forwarded to the leaves' parents; atomic events that do not match these event queries are not kept. Inner nodes process composite events they receive from their child nodes following the basic pattern:

1. try to build composite events by pairing up stored with newly detected events; this is done according to the semantics of the XChange construct the inner node implements,
2. update its storage by adding newly detected events that might be needed in future evaluation steps,
3. forward the events built in (1) to the parent node.

For exemplifying the evaluation algorithm for XChange composite event queries and ease its understanding, the binary conjunction (`and` construct) of event queries is chosen. Figure 5.1 sketches an implementation for the evaluation of a (binary) `and` inner node in java-like pseudocode (the figure is taken from [18]). Upon reception of a new events (event *a* in the example), the left and right child are evaluated to forward *newL* and/or *newR* (tuples corresponding to newly received event(s)) to their parent node *n* (the `and` operator node). The storage of n consists of lists *storageL* and *storageR* of events received from its left and right child nodes from previous evaluations. The current evaluation at n does the following:

1. compute all composite events from *storageL*, *storageR*, *newL*, and *newR*. That is, try to 'marry' each event from the left child with an event from the right child where at least one of them is a newly detected event (i.e. try to pair (s_L, Σ_L) with (s_R, Σ_R), where $((s_L, \Sigma_L), (s_R, \Sigma_R)) \in newL \times newR \cup newL \times storageR \cup storageL \times newR$). Two events (tuples) can only 'get married' if they agree on a suitable common substitution set Σ; as result of this pairing, a composite event $(s_L \cup s_R, \Sigma)$ corresponding to a conjunction is obtained. Σ is the maximal substitution set that assigns values to all variables with at least one defining occurrence in the child nodes and assigns same substitutions for variables occurring in both child nodes (it is assumed that Σ_L and Σ_R are undefined for other variables than the ones occurring in the left and right child, respectively). Σ is computed as a kind of natural join between Σ_L and Σ_R: $\Sigma := \Sigma_L \bowtie \Sigma_R$.

2. update *storageL* and *storageR* for future evaluations. As all detected events might be needed in future evaluations, the new storage of n is computed as

   storageL = storageL ∪ newL
   storageR = storageR ∪ newR

An example of evaluating a concrete conjunction event query on a concrete event stream in XChange is given in [18]. Recall now the Example 5.5 of an `andthen` event query with partial specification; the evaluation of such an event query follows the same lines as for the `and` operator described above, but considering the semantics of `andthen` (answers to $a\{\{var\,X\}\}$ and $b\{\{var\,X\}\}$ need to come in sequence). The middle child node (represented by $*$) has a storage in form of a simple list of atomic events. From these events, only those occurring between the answers to $a\{\{var\,X\}\}$ and $b\{\{var\,X\}\}$ are in the event sequence the operator node `andthen` forwards to its parent; the middle child node does not influence the substitution sets of the answers to `andthen`.

For evaluating XChange event queries, negation (in evaluating exclusion event queries) and existential quantified variables in composite event queries (variables in `any` specification inside e.g. a `times` construct) need a special treatment. For dealing with negation of events, the notion of substitution sets has been extended; besides substitutions that *assign* values to variables, substitutions that *forbid* variables to have certain values are used. For dealing with existential variables, the substitution sets to be joined (when trying to 'marry' events) need not agree on *all* common variables (as is the case for conjunction); substitution sets need to agree on all common variables *except* the existential quantified ones. In both cases, the join of substitution sets for determining answers to be forwarded is computed accordingly.

The algorithm presented for evaluating XChange composite event queries can yield *duplicate answers*; this situation can occur rarely; however, the evaluation algorithm would not be correct with respect to the declarative semantics if duplicate elimination would not be performed. A concrete example event query and event stream for which duplicate answers are obtained is given in [76], Section 7.3.6, page 95. For reasons of simplicity, the evaluation algorithm is not modified and duplicate elimination is performed after event query evaluation. It suffices to eliminate duplicate answers at the root of the operator tree.

Deletion of Events The operator tree is traversed in a *bottom-up manner* for detecting answers to composite event queries (by pushing events towards the root of the operator tree, and by adding events to the storage of the inner nodes); it is traversed in a *top-down manner* to remove events from inner nodes' storage if their lifespan has expired. To determine if the lifespan of an event has expired or not, the current time and the bound(s) of the event lifespan are used. XChange legal event queries always pose (finite) bound(s) on events' lifespan: temporal restriction and the `during FiniteTimeInterval` operator nodes (representing the corresponding XChange constructs) bound the lifespan of all events in their subtrees in the operator tree.

The deletion of events from the storage of the inner nodes of an operator tree begins at the root node. The idea is to use a time interval $[min..max]$ for determining the events that are not needed anymore (i.e. their lifespan has expired and they will not yield other answers to the whole event query representing the operator tree); this time interval is modified by traversing the operator tree. Not all nodes do update the time interval, only temporal restriction and `during FiniteTimeInterval` operator nodes influence $[min..max]$.

The initial time interval when starting the event deletion is $[min..max] = [-\infty..ct]$, where ct is the current time. As the operator tree always corresponds to a legal event query, the root node (representing an `in`, `before`, `within`, or `during` construct, cf. Section 4.4.4) is the first operator node that restricts (updates) the $[min..max]$ interval to $[min_u..max_u]$. How the interval is modified depends on the construct the operator node (regardless of root or inner node) implements:

- an `in` $[t_1..t_2]$ operator node imposes a temporal restriction to $[t_1..t_2]$ for all its subtrees; it updates the current $[min..max]$ interval to $[min_u..max_u] = [min..max] \cap [t_1..t_2]$;
- a `before` t operator node imposes a temporal restriction to $[-\infty..t]$ for all its subtrees; it updates the current $[min..max]$ interval to $[min_u..max_u] = [min..max] \cap [-\infty..t]$;
- a `within` w operator node restricts events (s,Σ) to those satisfying $end(s) - begin(s) \leq w$, that is $[begin(s)..end(s)] \subseteq [(ct-w)..ct]$; the operator node updates the current $[min..max]$ interval to $[min_u..max_u] = [min..max] \cap [(ct-w)..ct]$;
- a `during` $[t_1..t_2]$ operator node imposes a temporal restriction to $[t_1..t_2]$ for all its subtrees; it updates the current $[min..max]$ interval to $[min_u..max_u] = [min..max] \cap [t_1..t_2]$.

The whole operator tree is traversed from the root to the leaves; for removing events from nodes' storage the following are performed: The current time interval $[min..max]$ is updated to $[min_u..max_u]$ whenever a temporal restriction or `during` operator node is encountered (this is done as explained above). For every composite event (s,Σ) stored in the current node's storage test $[begin(s)..end(s)] \subseteq [min_u..max_u]$; delete (s,Σ) if this isn't the case. The subtrees of the current node are traversed with $[min_u..max_u]$ in the same manner.

Note that event deletion in XChange depends only on the current time and the composite event query; deleting events in the operator tree is independent on the data events carry with.

The algorithm presented for evaluating XChange event queries is amenable to optimisations. The focus of this thesis is not on developing efficient techniques for evaluating queries against volatile data; however, this is an interesting research issue. Some ideas for optimising the evaluation of XChange event queries are given in [76], Section 7.5, pages 98 -102.

The idea to prove *correctness* of the incremental algorithm w.r.t. the declarative semantics (defined in 5.1.1) is by dividing the problem into two: Forget that the algorithm is incremental and stores events; to detect an event at a time point t pretend that all incoming events are processed in one single evaluation. Then prove that the operator tree will always have stored the right events, that is, at time point t it stores all events that can be constituting part of a composite event with occurrence time t or later. This requires checking that in the bottom-up data flow all needed events are stored and that in the event deletion events that are still needed are not deleted.

5.2.2 Evaluation of Web Queries: Basic Ideas

Evaluating the Web queries given as 'condition part' of XChange reactive rules presupposes evaluating Xcerpt queries against specified resources and chaining of Xcerpt rules so as to evaluate Xcerpt queries against views constructed by means of Xcerpt rules. The operational semantics of the query language Xcerpt is defined in [125], Chapter 8. The underlying ideas of the semantics are shortly described in this section.

An algorithm is defined for evaluating Xcerpt programs based on two parts: an algorithm called *simulation unification* is given and a *backward chaining* algorithm that uses simulation unification. The evaluation is based on a simple constraint solver that applies simplification rules to a constraint store consisting of conjunctions and disjunctions of constraints. An example of a constraint is a *simulation constraint* expressing e.g. possible bindings for a variable. The constraint store yields bindings for the variables occurring in Xcerpt query and construct terms.

Simulation unification takes two terms and returns a set of variable substitutions (called *simulation unifier*) such that their applications to the terms make them simulate one into another. The underlying ideas of simulation between two terms have been presented in Section 2.4.2. A proof is given for the soundness and completeness of the simulation unification algorithm.

The backward chaining algorithm used in evaluating Xcerpt programs is inspired by the SLD resolution calculus used in logic programming [106]. It is shown that the algorithm is sound with respect to the fixpoint semantics developed for Xcerpt; also a (weak) completeness of the algorithm (is complete in cases where the algorithm terminates) is proved. Criteria for termination are also described in [125].

5.2.3 Execution of Updates

This section discusses the execution of XChange updates. It considers first some conceivable approaches for executing updates on Web data (updates specified by means of an update language). Then it recalls the model for updating data on the Web and the update operations considered in this work. The section ends by presenting the approach taken in XChange and already mentioned in Section 5.1.3; the underlying ideas of the taken approach are given and explained in detailed steps through a simple example of an XChange update.

Executing Updates to Web Data Different approaches are conceivable for executing the update operations specified by means of an update language for Web data; they are determined by different conditions or criteria that need to be taken into account. First, updating data depends on the representation formalism and the storage of the data to be modified. One can find data on the Web (as Web resources data) represented in a multitude of formalisms (such as HTML, XML, RDF, or relational databases) and data storage (e.g. XML data can be stored as native XML documents, in relational databases such as Tamino [1], or in object-oriented databases). Clearly, the representation of data is more important for the language constructs and the storage of data for the execution of updates specified through these constructs.

Let's now consider another 'dimension' of update execution. Updates can be performed

(i) *on secondary storage*, meaning that the specified update operations are executed directly (data need not be loaded entirely in memory) on the data to be updated, or

(ii) *in memory*, meaning that the data to be updated is loaded in the memory where it is modified, the data after the update need to be 'placed instead' of the old (initial) data. Using this approach, the data after the update can be constructed in memory or the update operations can be performed on the internal representation of data (e.g. on the DOM representation when updating XML data). However, this approach is not suitable for updating large documents.

Also, an update language can have proprietary update execution abilities, or transformation rules can be provided for the language constructs into existing 'update management' means. For the latter case, a mapping between the update constructs of the language and update or construction constructs of another language are provided; evaluating or executing the obtained programs yields the same effect as if a proprietary update processor is available. For example, update operations on XML documents can be mapped into SQL update operations that work on an XML (relational) database; such XML to SQL mappings eliminate the need to understand the database structure.

Efficiency issues can play an important role when updating data on the Web; however, there are few proposals for efficient execution of updates on Web data having (native) XML storage. For example, finding the least expensive sequence of operations to transform an initial document (before any update is performed) in the final one (the document after the specified update are performed) poses interesting research problems.

XChange Updates on the Web An XChange program is located at one (XChange-aware) Web site and contains rules specifying ordered or unordered conjunctions and/or disjunctions of updates. Updates are specified as 'action part' of XChange reactive rules; they are performed after the successful evaluations of the other parts (event query, Web query) of the rules. Thus, the substitution set $\Sigma_u = \Sigma_{eq} \bowtie \Sigma_{wq}$ is used in performing the specified updates, where Σ_{eq} and Σ_{wq} are the substitution sets obtained from evaluating event query part and Web query part, respectively.

XChange updates express how data found at one or more Web resources are to be modified, i.e. how persistent data is to modified. These Web resources are either local or remote. Updates to local Web resources are executed by the language processor at the Web site. Updates to remote Web resources are *not* executed by the processor of the Web site where the update has been specified; instead updates to remote data are *update requests* to the Web sites where the data to be modified is stored. A Web site receiving an update request can try to execute the update or decide not to execute the requested update. This approach is consistent with the local control of XChange programs.

An XChange elementary update consists of a resource specification (the resources to be updated) and an update term (a pattern for the data to be modified augmented with update operations). The subjacent query term of an update term is the underlying query pattern of the respective update term. Consider an elementary update u specifying modifications of data term d_u through update term t_u whose subjacent query term is s_{t_u}; a premise for a successful execution of the update operations of u is the satisfaction of the condition $s_{t_u} \preceq d_u$. In other words, the query s_{t_u} needs to evaluate successfully against d_u for performing the given update operations on d_u. The evaluation of the subjacent query term of an update term against the given data to be modified can represent a 'pre-update operations execution' step for determining whether to (try to) execute the update operations on these data or not. However, performance results need to be compared for executing updates with and without subjacent query term evaluation for determining which technique is more suitable and less expensive.

XChange update operations specify insertions, deletions, or replacements of data for tree-like Web data. Executing an update operation

- `insert ConstructTerm` implies the construction of a data term using `ConstructTerm` and the variable substitutions obtained from the other parts of the rule and the evaluation of subjacent query term; the construction follows closely that of Xcerpt [125]. Where the constructed data term is inserted is given by the position of the insertion operation inside the subjacent query term.

- `delete QueryTerm` deletes all terms matching the `QueryTerm`; all subterms of these terms are deleted.

- `QueryTerm replaceby ConstructTerm` replaces all terms matching the `QueryTerm` with a data term constructed with `ConstructTerm`.

The other constructs for XChange update operations are executed conform their meaning (that have been introduced informally in Section 4.6), for example by inserting at a given position in the document when a position is specified.

An XChange complex update specifies ordered or unordered conjunctions or disjunctions of (elementary or complex) updates. Executing a complex update

- `and` $[U_1, \ldots, U_n]$ means executing all U_i, $1 \leq i \leq n$ in the given order so as to use a substitution set for the variables obtained by executing U_i in the subsequent updates U_j, $i+1 \leq j \leq n$.

- `and` $\{U_1, \ldots, U_n\}$ means executing all U_i, $1 \leq i \leq n$ regardless of the execution order (unordered updates can be executed in parallel).

- `or` $[U_1, \ldots, U_n]$ means executing one of the U_i, $1 \leq i \leq n$ by trying to execute the updates in the given order and stop after the first successful execution of a specified update.

- `or` $\{U_1, \ldots, U_n\}$ means executing one of the U_i, $1 \leq i \leq n$; the processor can pick freely the update to be executed from the given ones.

The execution of complex updates is a kind of controlled execution of two or more elementary or complex updates; the building block consists in executing XChange elementary updates. The approach taken in XChange for executing elementary updates is discussed in the following.

Updates through Construction The approach taken for executing XChange updates is an in memory execution of updates where the elementary updates are executed by constructing the data after the update. Mappings between XChange elementary updates and Xcerpt goals are provided. Data to be updated has a tree-like representation (e.g. XML, or RDF data) stored as XML documents. The approach taken in executing XChange updates is not the most efficient alternative to update execution; methods for more efficient execution of updates on the Web are subject to further research.

The idea of executing XChange updates is based on an interesting observation, namely that the XChange updates are an *elegant* and *easy* way for specifying data modifications as an alternative to an intentional specification, i.e. constructing the data after the update by means of deductive rules. That is, for each XChange elementary update a corresponding Xcerpt goal exists, such that evaluating the Xcerpt goal has the same effect as if the respective update operations were executed directly on the data. The substitution set obtained from evaluating the other parts of the rule having the elementary update as action or from executing other specified updates in the action part is used in evaluating the Xcerpt goal. The data term constructed by evaluating the Xcerpt goal is the data term after the update; this modified data term overwrites the initial data (the data before the update).

Given an XChange elementary update u, a corresponding Xcerpt goal G of the form $ConstructTerm \leftarrow_g QueryTerm$ is constructed by taking the structure of the subjacent query term and the update operations of u into account. This transformation poses the following challenges:

(i) *partial patterns* in u do not offer knowledge about all subterms of terms in the data to be modified; means are needed for determining whether a term has other subterms than those specified in the query pattern, and for gathering all these subterms if they exist so as not to loose data through construction.

(ii) the *original order* of the terms in the data to be modified needs to be maintained.

(iii) the *semantics* of XChange update operations needs to be mirrored by the constructed Xcerpt goals.

(iv) the *position specification* in insertion operations express insertion of data at the given position; means are needed so as to assure that the modified data contains the inserted data at the given position.

Rewriting rules have been recognised for rewriting an XChange elementary update specification into a corresponding Xcerpt goal specification. These rules are applied recursively on the structure of the given update term to obtain a tuple $(ConstructTerm, QueryTerm)$ consisting of the construct and query part of an Xcerpt goal. The resources given in the elementary update are 'forwarded' to the query and construct part of the Xcerpt goal. The rewriting rules comprise the following solutions to the problems touched on above:

(i) partial patterns imply the use of Xcerpt's construct `optional` before a fresh variable in the goal's query and gathering of all these in its construct term (so as to ensure that no data is lost along the construction way);

(ii), (iv) the desired order of the subterms in the modified data term is assured by combining the use of ordered patterns with Xcerpt's ordering of terms based on their position.

(iii) the desired effect of XChange update operations is achieved by a careful development of the rewriting rules based on the fact that XChange update terms consist of Xcerpt terms (query terms and construct terms) and update constructs.

For proof-of-concept purposes, rewriting rules for transforming XChange elementary updates into corresponding Xcerpt goals have been implemented; they cover a representative "class" of XChange update terms and are given in Appendix B. Ongoing work concerns testing the implemented rules to determine to which extent all possible XChange update patterns are covered and to reveal details that have been possibly neglected.

The following example explains the transformation steps that are needed in order to go from an XChange elementary update to a corresponding Xcerpt goal.

Example 5.6 (Flight Reservation Specified as Deductive Rule)
Recall the Example 4.57 specifying an XChange transaction rule for booking another flight as reaction upon a flight cancellation. After evaluating the event query and Web query parts, the specified action is to be performed. Its specification follows:

```
in { resource { "http://airline.com/reservations/" },
     reservations {{
        insert reservation { var F, name { "Christina Smith" } }
       }}
    }
```

For constructing a corresponding Xcerpt goal for the above given XChange elementary update simple steps need to be taken by paying attention to the structure of the update term. The resource `http://-airline.com/reservations/` to be modified is treated in a straightforward manner: the query part of the goal queries it and the construct part of the goal specifies it as the output resource (where the data after the update should be 'put').

The subjacent query term of the given update term is

```
reservations {{  }}
```

which is transformed by applying the rewriting rules in a query term to be used in the query part of the goal and a construct term to be used in the construct part of the goal. The query term is obtained by adding a pattern that matches with the subterms of the `reservations` term; the subterm and its position are to be bound to the variables `Child` and `CPos`, respectively. The `optional` construct is used because there is no knowledge about the existence of other made reservations (or subterms of other kind) at `http://-airline.com/reservations/`. Thus, the following query term is obtained

```
reservations {{
   optional position var CPos  var Child
}}
```

Let's turn attention to the construction of the goal's construct term. The partial specification turns into total ordered specification so as to keep the order of the subterms as in the initial data term (i.e. the data term before the update is performed). All subterms of the root `reservations` in the initial data term are gathered by means of the construct `all` (so as not to loose information) and ordered by their position in the initial data term (for keeping the initial order). After these steps, the construct term looks like

```
reservations [
    all optional var Child order by [ var CPos ]
]
```

However, using this construct term one just constructs the initial data term without taking the insertion update into consideration. One more step needs to be made; the construct term specified in the insertion operation (i.e. after the `insert` keyword) is a pattern used in the goal's construct term for constructing a new subterm of `reservations`. Thus, the goal's construct term is complete as:

```
reservations [
    all optional var Child order by [ var CPos ],
    reservation { var F, name { "Christina Smith" } }
]
```

The corresponding Xcerpt goal built by making the described transformation steps to the elementary update given at the beginning of this example is the following:

```
GOAL
  out { resource { "http://airline.com/reservations/" },
        reservations [
             all optional var Child order by [ var CPos ],
             reservation { var F, name { "Christina Smith" } }
        ]
      }
FROM
  in { resource { "http://airline.com/reservations/" },
       reservations {{
            optional position var CPos  var Child
        }}
     }
END
```

The desired booking is realised (i.e. the insertion update is executed) by evaluating the above given Xcerpt goal. The reservation for Christina Smith is to be found as the last subterm of the `reservations` term in the data at `http://airline.com/reservations/`.

The previously given example of an XChange elementary update is a simple one that keeps its simplicity in the transformation process and the corresponding Xcerpt goal specification. However, more complex XChange updates are still easy to be transformed (the rewriting rules are applied recursively on the structure of the update terms) but lack clear, simple specifications of the corresponding goal for programmers. As the whole transformation remains hidden and can be realised by automatic means, programmers need just to use the elegant XChange update operations.

CHAPTER

SIX

Use Cases

Developing use cases for a programming language aims at revealing the strengths and limits of the language. When designing a language, use cases from intended application domains are first developed so as to identify requirements the language should fulfil. During the process in which the language gets mature, use cases are developed so as to bear evidence of the practicability of language constructs. The same approach has been taken in developing the language XChange whose constructs have been introduced in Chapter 4.

The *World Wide Web Consortium* (W3C) [3] has published a set of use cases for different kinds of languages for the Web, such as XML or RDF query languages; when developing a new Web language of one of these kinds, application scenarios that correspond to the ones described by the W3C are means for showing the abilities of the language by comparing it to existing languages. Though, at moment there is no such W3C document describing use cases for a reactive language. No use cases or classes thereof for reactivity on the Web exist that can be taken as reference for developing XChange use cases. The application scenarios developed for XChange and implemented in part in this chapter have been motivated by their use in real-life situations. In general, there is no use case of reasonable size whose implementation uses all the constructs a language offers; thus, besides application scenarios for travel planning and support, this chapter touches also other two application areas where the work on XChange use cases seems promising.

This chapter is structured as follows: Section 6.1 presents (part of) an implementation in XChange of the use case *Travel organisation* whose explanation has been given in Section 1.2.1. Section 6.2 gives flavour of two XChange use cases, one aiming at showing that XChange can also be employed for Semantic Web applications and one at showing that XChange is suitable for implementing *business rules*.

6.1 Travel Organisation

Travel organisation is an application of Web-based reactive travel planning and support. In realising such an application two subtasks need to be considered: 1. initial planning and 2. reacting to happenings that influence the plan. These subtasks have been explained in more detail in Section 1.2.1. *Note* that the task of travel planning and support does not involve a single system. Instead, in order to save time and perfectly manage trips, several systems must cooperate to realise 1. and 2., e.g. flight and train schedules, passenger notification system, hotel reservation service.

For exemplifying the task of organising travels, consider that Mrs. Smith uses a travel organiser that plans her trips and reacts to happenings that could influence her schedule. Exemplary application scenarios have been implemented that fit into the *Travel organisation* setting; they have as common story-line the organisation of Mrs. Smith's vacation in Provence, France. Mrs. Smith wants to visit a couple of cities (Lyon, Orange, Arles, Nîmes, and Marseilles). However, for space and simplicity reasons the scenarios of this section are restricted to a single city to visit, Lyon; flavour of extending the given implementation to the scenarios of Section 1.2.1 are given throughout the section. Mrs. Smith's vacation is planned for 5th to 20th of March 2005. The scenario of Section 6.1.1 corresponds to the task of *initial planning*, while the scenario of Section 6.1.2 corresponds to the task of *adapting plans to changes*. Unless otherwise stated, all XChange rules presented in the following sections implement a single system – the travel organiser. *Note* that the order in which the travel organiser's rules are presented here is not of importance (cf. Section 5.2).

6.1.1 Initial Planning Scenario

Planning Mrs. Smith's vacation implies gathering e.g. information about hotels, flights, trains, corresponding prices; to this aim, deductive rules (i.e. Xcerpt rules) can be employed. Though, for taking notifications such as offered flight discounts into considerations, reactive rules are needed.

For gathering information about hotels located in the cities Mrs. Smith would like to visit, (possibly) different Web resources' data need to be queried. Some Web sites offer information on more than one city, while others are restricted to hotels in a specific city. Excerpts of data from two such Web sites follow; at `http://hrs.net` one can find information about hotels in e.g. Lyon and Arles, at `http://h-lyon.fr` just information about hotels in Lyon. *Note* that besides the information about the city where a particular hotel is located, no other location data is given; these kind of data have been abstracted as XChange does not yet have the ability to reason about location data. Integrating a language capable of location data representation and reasoning into XChange is one possible direction for further work (cf. Section 7.2).

```
At http://hrs.net

accommodation {
  service{"http://hrs.net/res/"},
  currency {"EUR"},
  hotels {
    city {"Lyon"},
    hotel {
      name {"Princesse Isabelle"},
      phone {"+33 123 456"},
      category {"3 stars"},
      price-per-room {"112"}
    },
    hotel {
      name {"Corail"},
      phone {"+ 33 123 789"},
      category {"2 stars"},
      price-per-room {"55"},
      no-pets { }
    },
    ...
  },
  hotels {
    city {"Arles"},
    hotel {...},
    hotel {...},
    ...
  },
  ...
}
```

```
At http://h-lyon.fr

logement {
  monnaie {"EUR"},
  ville {"Lyon"},
  hotel {
    nom {"Unic"},
    telephone {"+33 123 123"},
    classe {"3 etoiles"},
    prix {"112"}
  },
  hotel {
    nom {"Corail"},
    telephone {"+33 123 789"},
    classe {"2 etoiles"},
    prix {"55"}
  },
  hotel {
    nom {"Istria"},
    telephone {"+33 123 789"},
    classe {"3 etoiles"},
    prix {"65"}
  },
  ...
  service{"http://h-lyon.fr/res/"}
}
```

The following deductive rule queries data found at Web resources `http://hrs.net` and `http://h-lyon.fr` and constructs a view over the hotel data by gathering information about all listed hotels in Lyon. The constructed data term contains a list of hotels ordered by their price per room. For each hotel the service where the hotel can be booked needs also to be provided. *Note* that same kind of rules are used for gathering information for the overnight stays in the other cities to visit.

```
CONSTRUCT
```

```
 hotel-info [
    city {"Lyon"},
    all hotel { service { var Service },
                name { var Name },
                price { var Price },
                phone { var Phone } } order by ascending [ var Price ]
   ]
FROM
 or {
    in { resource { "http://hrs.net" },
         accommodation {{
            service { var Service },
            hotels {{
               city { "Lyon" },
               desc hotel {{
                       name { var Name }
                       price-per-room { var Price },
                       phone { var Phone } }} }}
         }}
     },
     in { resource {"http://h-lyon.fr"},
          logement {{
             hotel {{
                nom { var Name },
                telephone { var Phone },
                prix { var Price }
             }}
             service { var Service },
           }}
     }
 } where var Price < 70
END
```

So as to query hotel data in an uniform manner when making the necessary booking, a view over the hotel data in the cities of interest is constructed by means of the following deductive rule. The rule queries the result of the other existing rules gathering hotel information (i.e. rules like the one given above).

```
CONSTRUCT
 hotels {
    all view [
           var City,
           all var Hotel
        ]
 }
FROM
   hotel-info [[
      var City → city {{ }},
      var Hotel → hotel {{ }}
   ]]
END
```

A similar approach is taken for gathering information about flights. Offers of different airlines are queried for finding Munich – Lyon connections: deductive rules in the style of those used for gathering hotel information are used, for space reasons they are not given here (for more use cases for the Web query language Xcerpt used for specifying deductive rules in XChange, see [98, 125, 33]). Data to be queried contains information about e.g. flight number, departure and arrival airports, class, price having the following possible structure:

```
flights {
  airline { "AI" },
  currency { "EUR" },
  flight {
    number { "AI2000" },
    from { "Munich" },
    to { "Lyon" },
    date { "2005-03-05" },
    departure-time { "6:15" },
    arrival-time { "7:50" },
    class { "economy" },
    price { "55" }
   },
  flight {
    number { "AI2011" },
    from { "Lyon" },
    to { "Munich" },
    date { "2005-03-20" },
    departure-time { "10:30" },
    arrival-time { "12:00" },
    class { "economy" },
    price { "75" }
   },
  flight {
    number { "AI2021" },
    from { "Lyon" },
    to { "Munich" },
    date { "2005-03-20" },
    departure-time { "17:30" },
    arrival-time { "19:00" },
    class { "economy" },
    price { "80" }
   },
  ...
}
```

Views over flight data where the flight to Lyon and back costs less than 400 Euro can be constructed as for hotel data; here, recursion can play an important role for determining connections that are not direct. However, this scenario considers just direct connections from Munich to Lyon and return:

```
CONSTRUCT
 connection {
    all direct [
         outward { all var Outward },
         return { all var Return }
        ]
 }
FROM
   flights {{
      var Outward → flight {{ date {"2005-03-05"},
                         from {"Munich"}, to {"Lyon"}
                         without via {{ }}
            }}
      var Return → flight {{ date {"2005-03-20"},
                         from {"Lyon"}, to {"Munich"}
                         without via {{ }}
            }}
   }}
END
```

Assume that the airlines notify the persons who have already been their passengers about discounts they are offering. The discount information is sent to the travel organisers as event messages. The following XChange event-raising rule is part of an XChange reactive program running at an airline's Web node:

```
RAISE
  all xchange:event {
         xchange:recipient { var To },
         news {
            subject {"Flight discounts"},
            discount {
                   airline { "AI" },
                   from {"2005-02-25"}, until {"2005-03-25"},
                   percent {"10"}  }
         }
      }
FROM
  in { resource { "file:passengers.xml" },
       desc passenger-data {{
                year { "2005" },
                passenger {{
                   organiser { var To} }}
        }}
  }
END
```

In order to be able to determine the most advantageous flight Munich – Lyon and return, the travel organiser makes the discount information persistent. The next XChange reactive rule is triggered by the event-raising rule given above, i.e. the rule fires upon reception of the discount event message.

```
TRANSACTION
  in { resource {"file:flight-discounts.xml"},
       discounts {{
          insert var D
       }}
  }
ON
  xchange:event {{
     news {{
         subject {"Flight discounts"},
         var D → discount {{
                   airline {{ }},
                   from {{}}, until {{}},
                   percent {{}}
             }}
    }}
  }}
END
```

The deductive rule that determines the cheapest direct flight(s) from Munich to Lyon and back query the data at `flight-discounts.xml` and the view constructed with the previously given deductive rule; the rule is not given here as it does not make use of any other constructs than the ones presented so far.

The travel organiser uses an XChange transaction rule for booking a flight from Munich to Lyon and return and, after a successful flight booking, booking overnight stays in Lyon. Also, the travel organiser stores as persistent (in file `trips.xml`) information about the made arrangements, so as to (i) be able to react to events that can affect these arrangements, and (ii) rate e.g. hotels based on Mrs. Smith's input for influencing further arrangements.

```
TRANSACTION
  and [
     in { resource { var Agency},
          desc reservations {{
              insert reservation {
                     var F, name {"Christina Smith"}
              }
          }}
     },
     in { resource { var Service },
          accommodation {{
             insert reservation {
                     var H, name {"Christina Smith"},
                     from {"2005-03-05"}, until {"2005-03-08"}
             }
          }}
     },
     in { resource { "file:trips.xml" },
          trip [[
            at 1 insert vacation {
                     location { "Provence" },
                     flight { var F,
                              service { var Agency } },
                     var Hotel  }
          ]]
     }
  ]
FROM
   and {
     best-flights [[
       Position 1 choice {
             var F → connection {{ }},
             reserve-at { var Agency }
       }
     ]],
     hotels {{
       view [[ city {"Lyon"},
               Position 2 var Hotel → hotel {{
                              service { var Service },
                              name { var H }  }}
        ]]
     }}
   }
END
```

After successful execution of the transaction given previously, the file `trips.xml` looks as follows:

```
trip [
  vacation {
     location { "Provence" },
     flight {
        connection { direct {
            outward {
               flight {
                  date {"2005-03-05"},
                  from {"Munich"}, to {"Lyon"},
                  ...
               }
            },
            return {
               flight {
                  date {"2005-03-20"},
                  departure-time {"17:30"},
                  arrival-time {"19:00"},
                  ...
               }
            }
        }  },
        service {"http://ai.com/res/"}
     },
     hotel {
        service {"http://h-lyon.fr/res/"},
        name {"Istria"},
        phone {"+33 123 789"},
        price {"65"}
     }
  },
  business-trip { ... },
  ...
]
```

Note that for the case of visiting more than one city, the file `trips.xml` needs to be adapted so as to store for each period of time the corresponding hotel booked for Mrs. Smith.

The next XChange reactive rule is dedicated to planning entertainment. As reaction to the reception of an exhibition notification and a forecast notification, the travel organiser orders a ticket for Mrs. Smith and notifies a friend of her about the exhibition of G. Barthouil.

```
TRANSACTION
  and {
     in { resource {"http://artactif.com/tickets.xml"},
          desc exhibition {{
               insert ticket-order {
                     var P, var L, name {"Christina Smith"}
               }
          }}
     },
     xchange:event {
        xchange:recipient {"http://organiser.fr/˜Ramona"},
        var Notif
     }
  }
ON
and {
  var Notif → xchange:event {{
       xchange:sender {"http://artactif.com"},
       exhibition {{ var P → painter {"G. Barthouil"},
                     var L → location {"Lyon"},
                     time-interval { var TI }
                  }}
   }},
  xchange:event {{
    xchange:sender {"http://weather.com"},
    forecast { date { var D }, city {"Lyon"},
               info {"It's going to rain."} }
   }}
 } before 2005-03-01T11:15:00
 where var D included-in var TI
        and [2005-03-05..2005-03-08] included-in var TI
END
```

In the same manner (using XChange transaction rules as the ones presented) flight and train tickets can be booked for arranging the trip of Mrs. Smith so as to visit other cities (Orange, Arles, Nîmes, and Marseilles) too. The successful execution of the transaction booking a flight to and a corresponding hotel in Lyon triggers rules for arranging appointments with Mrs. Smith's friends.

```
RAISE
  all xchange:event {
         xchange:recipient { var To },
         subject { "Can we meet?" },
         info { "Dear"+var N+"I'm in Lyon between 5th and 8th of March" }
  }
ON
   xchange:event {
      xchange:type { "commit" },
      conjunction {
         flight {{ }},
         hotel {{ }}
      }
  }
FROM
  in { resource {"file:address-book.xml "},
       addresses {{
         friends {{
           info {{
              city { "Lyon" },
              name { var N },
              organiser { var To }
           }}
         }}
       }}
  }
END
```

Note that for arranging a trip (gathering information, make reservations and arrangements) deductive rules and transaction rules that are not event-driven are not sufficing; transactions and raising of events as reactions to (local and remote) events need also be employed.

6.1.2 Adapting to Changes Scenario

The XChange rules given in the previous section implement part of the abilities of Mrs. Smith's travel organiser, which are needed for making all the arrangements for an initial plan. Besides these rules, the travel organiser has rules dedicated to adapting already made plans to happenings that occur during Mrs. Smith's vacation. XChange event-raising rules and transaction rules are employed for detecting and reacting e.g. to important changes at office, to family-related events, to hotel over-booking, to train or flight delays, or to flight cancellations. It can be noticed that the number of classes of events and the number of possible reactions to them is quite big. Though, this section gives a few exemplary XChange reactive rules that show XChange's abilities to elegantly implement the task of adapting plans to changes.

The next XChange event-raising rule is fired during Mrs. Smith vacation if she receives at least three important messages during a vacation day. *Note* here the use of the temporal type `day` that can be defined using the calendar and temporal type system CaTTS [51] like

```
type day = aggregate 24 hour @ hour(1) ;
```

where type `hour` is similarly defined. Using the next XChange rule, the travel organiser detects important messages, looks for the organiser's address of the sender, and sends him/her the phone number of Mrs. Smith's hotel. *Note* that if Mrs. Smith does not have the address of the messages' sender, the travel organiser does not reveal the hotel where its owner is staying at.

```
RAISE
   xchange:event {
      xchange:recipient { var From },
      contact {
         text {"You might want to call Mrs. Smith"},
         var Name, var Phone
      }
   }
ON
   times atleast 3 any var S {
         xchange:event {{
            xchange:sender { var From },
            important-message {{
               subject { var S }
            }}
         }}
   } during day
FROM
  and {
     in { resource {"file:address-book.xml "},
         desc organiser { var From }
     },
     in { resource {"file:trips.xml"},
          trip [[
            Position 1 vacation {{
                 desc hotel {{
                       var Name → name {{ }},
                       var Phone → phone {{ }}
                    }}
            }}
          ]]
     }
  }
END
```

The next XChange reactive rule deals with delays of flights Mrs. Smith travels with. As the airline might send several delay notifications for a flight, the rule detects the last notification that occurred during the first signalling of delays and the notifications announcing the boarding time. The detected composite event triggers an event message to be sent to one of Mrs. Smith's friends so as to pick her up.

```
RAISE
   xchange:event {
      xchange:recipient { "http://organiser.de/~Simona" },
      var AT
   }
ON
  last {
     xchange:event {{
       xchange:sender { var Service },
       delay-notification {{
          var Flight,
          expected-departure-time { var DT },
          var AT → expected-arrival-time {{ }}
       }}
     }}
   } during {
           andthen [
              withrank 1 {
                 xchange:event {{
                   xchange:sender { var Service },
                   delay {{ var Flight → number {{ }} }}
                 }}
              },
              xchange:event {{
                xchange:sender { var Service },
                boarding-time {{ var Flight, begin { var BT }  }}
              }}
          ]
  }
FROM
  in { resource {"file:trips.xml"},
       trip [[
         Position 1 vacation {{
           flight {{
              desc return {{ desc var Flight }},
              service { var Service }
             }}
         }}
       ]]
  }
END
```

The next XChange reactive rules deal with cancellation of flights Mrs. Smith travels with. On cancellation of Mrs. Smith flights where the airline grants and accommodation, the travel organiser announces her friend about the changes in her plan.

```
RAISE
   xchange:event {
      xchange:recipient { "http://organiser.de/˜Simona" },
      important {"Flight cancelled! I'm back on 21st of March!"}
   }
ON
   andthen [
      xchange:event {{
         xchange:sender { var Service },
         cancellation {{ var Flight → number {{ }} }}
      }},
       xchange:event {{
         xchange:sender { var Service },
         granted-accommodation {{ }}
      }}
   ] before 2005-03-21
```

```
FROM
  in { resource {"file:trips.xml"},
       trip [[
         Position 1 vacation {{
           flight {{
              desc return {{ desc var Flight }},
              service { var Service }
             }}
         }}
       ]]
  }
END
```

The following XChange transaction rule is used in case of flight cancellations where no accommodation is granted by the airline. The travel organiser looks for and books another return flight for Mrs. Smith.

```
TRANSACTION
  and [
     in { resource { var Agency},
          desc reservations {{
              insert reservation {
                     var Flight, name {"Christina Smith"}
             }
          }}
     },
     in { resource { "file:trips.xml" },
          trip [[
             Position 1 vacation {
                         location { "Provence" },
                         new-flight { var Flight,
                                      service { var Agency } }
             }
          ]]
     }
  ]
ON
   andthen [
      xchange:event {{
         xchange:sender { var Service },
         cancellation {{ var Nr → number {{ }} }}
      }},
      without { xchange:event {{
                   xchange:sender { var Service },
                   granted-accommodation {{ }}
                 }}
               } during [2005-03-05..2005-03-20T]
   ] before 2005-03-21
FROM
  and {
    in { resource {"file:trips.xml"},
         trip [[
           Position 1 vacation {{
             flight {{
                desc return {{ desc var Nr }},
                service { var Service }
               }}
           }}
         ]]
    },
   best-flights [[
     choice {{
        desc return {{
             var Flight → flight {{
                   without var Nr
             }}
          }},
         reserve-at { var Agency }
        }}
      ]]
  }
END
```

6.2 Flavour of Further Use Cases

Ongoing work in the XChange project focuses on the development of use cases for the language; it aims at revealing possible shotcomings of the language when developing Semantic Web apllications or modelling business processes. This section gives flavour of two further XChange use cases: a simple Semantic Web book store and a case study where workflows and business rules are employed.

6.2.1 E-Book Store – A Simple Semantic Web Scenario

XChange has been conceived not only for standard Web but also for Semantic Web applications. Such applications of XChange are presented and discussed in [43, 24]. It is crucial that relevant changes to data that has been used by a Semantic Web agent, e.g. in deciding which book to buy or what train to book, are consistently and rapidly propagated to all interested parties.

This scenario considers a Semantic Web-based book store that coordinates its activities based on clients orders and publishers' offers. The e-book store (an Amazon-like book store[1]) offers books that are presented to the clients as a collection of related Web pages (i.e. one or more Web sites); both for data presentation and for (internal) data processing, Semantic Web technologies are employed (such as ontologies that bring semantics into the data and allow for reasoning with data). The e-book store processes buy orders from its clients (implying e.g. order reception, updating store data on books), notifies its clients about books that are in particular class of books or other literary works and are available for buying, notifies clients about new published books (that are ordered just as reaction to clients' interests). It also makes book buy orders to publishers as reaction to updates to its store or to client orders. This section gives an excerpt of the data used by the e-book store and a couple of simple XChange rules for implementing tasks of an e-book store of such kind.

The data term of Figure 6.1 shows a small excerpt from a book database together with a sample ontology over novels and other literary works. Some of the concepts used are drawn from the "Friend of a Friend" (foaf) project[2]. The excerpt database is taken from [43, 24], where a more detailed use case for the Web and Semantic Web query language Xcerpt and for the reactive language XChange is given. *Note* that prefixes are used to abbreviate the URLs for properties.

The data contains two books. One is classified (via `rdf:type`) as a *Historical Novel* (defined in the sample ontology). The sample ontology is a conceptual hierarchy for classifying books and other literary works. The terms of the ontology are related by `rdfs:subClassOf`, meaning that, e.g., a *Historical Novel* is a kind of *Novel* that, in turn, is a kind of *Writing*.

[1] Amazon, `http://amazon.com`

[2] Foaf Project, `http://www.foaf-project.org/`

```
RDF {
  Historical_Novel {
    author {
      foaf:Person {
        foaf:name{"Colleen McCullough"}
      }
    },
    dc:title{"The First Man in Rome"}
  }

  Historical_Essay {
    author {
      foaf:Person {
        foaf:name { "Julius Caesar" }
      },
      foaf:Person {
        foaf:name { "Aulus Hirtius" }
      }
    },
    dc:title { "Bellum Civile" },
    translator {
      foaf:Person {
        foaf:name { "J. M. Carter" }
    } } }

  &author @ rdf:Property {
    rdfs:domain {
      ^&writing
    }
    rdfs:range {
      ^&foaf:Person
    }
  }
```

```
 &translator @ rdf:Property {
   rdfs:domain {
     ^&writing
   }
   rdfs:range {
     ^&foaf:Person
   }
 }

 &historical_novel @ rdfs:Class {
   rdfs:label { "Historical_Novel" },
   rdfs:subClassOf {
     &novel @ rdfs:Class {
       rdfs:label { "Novel" },
       rdfs:subClassOf {
         &writing @ rdfs:Class {
           rdfs:label { "Writing" }
         } } }
   }
   rdfs:subClassOf {
     &historical_essay @ rdfs:Class {
       rdfs:label { "Historical_Essay" }
       rdfs:subClassOf {
         &essay @ rdfs:Class {
           rdfs:label { "Essay" }
           rdfs:subClassOf {
             ^&writing
       } } } }
   }
 }
}
```

Figure 6.1: Excerpt of Book Database at `http://bookstore.com`

```
TRANSACTION
in {
  resource { "http://bookstore.com" },
  RDF {{
    insert Historical_Novel {
      dc:title { "Ein Kampf um Rom" }
      author {
        foaf:Person {
          foaf:name { "Felix Dahn" }
        }
      }
    }
  }}
}
END
```

Figure 6.2: Insertion of a New Historical Novel

```
RAISE
  xchange:event {
    xchange:recipient { "http://organiser.de/˜Smith" },
    new-book {
       type { var Type },
       all optional var Title
    }
  }
ON
  xchange:event {
      xchange:type {"update"},
      insertion {
         resource { "http://bookdealer.com" },
         term {
            var Type {{
               optional var Title → dc:title{{}}
            }}
          },
         parent { RDF {{ }} }
      }
  }
FROM
  subClassOf[
    rdfs:Class {{ rdfs:label{var Type} }},
    rdfs:Class {{ rdfs:label{"Essay"} }}
  ]
END
```

Figure 6.3: Notifying Mrs. Smith

The XChange transaction rule of Figure 6.2 is used for inserting a new novel titled `Ein Kampf um Rom` in the book database found at `http://bookstore.com`. Slight modifications of the rule allow for updating the book database with e.g. more than one book that are part of different classes of books.

Mrs. Smith is very interested in Essays and therefore wants to be notified about any new book that is classified as an Essay once it is added to the list of books managed by `http://bookstore.com`. By using the XChange event-raising rule of Figure 6.3, `http://bookstore.com` sends notifications triggered by insertions of books of type Essay.

Note that the 'condition part' of the rule does not query a given Web resource; instead it queries a view over data that is constructed by means of an Xcerpt rule, a rule that computes the transitive closure of `rdf:subClassOf` by using recursion. The Xcerpt rule is given in [43] as *Example 12* on page 14.

Extensive work on Semantic Web use cases for the reactive language XChange is planned so as to identify possible extensions to the language that are required by a reactive Semantic Web. Also, combining XChange with other Semantic Web technologies (so as to be able to reason with different kinds of data) needs to be investigated.

6.2.2 EU-Rent – Business Rules for Reactivity on the Web

At moment, use cases are developed that aim at showing to which extent the reactive language XChange is suitable for implementing *business rules*. The business rules approach has been employed with success in the industry for describing the business logic; thus, their practicability has been already proven.

"A business rule is a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business". (Business Rules Group[3]) For example, "an order with value greater than 500 Euro has no delivery charge" is a business rule.

Focus of the XChange use cases is the implementation of business rules and not their specification e.g. as a controlled English language or the validation of sets of business rules. However, not all business rules can be implemented; for example, "everyone in a construction area must wear safety helmet" can not be implemented by using programming languages. Though, the first given example of a business rule can be implemented (e.g. by using XChange as programming language).

Multiple classifications of business rules exist; the Rule Markup (RuleML) Initiative[4] considers three kinds of business rules: *integrity constraints*, *derivation rules* (also called deductive rules), and *reaction rules* (also called reactive rules). Integrity constraints can be implemented for example by means of schema languages or can be enforced by means of reactive rules. The fact that the language XChange has reactive rules (i.e. XChange event-raising rules and transaction rules) *and* deductive rules (i.e. Xcerpt rules) give reasons to claim that XChange is suitable for implementing (most kinds of) business rules. The developed use cases will bear evidence of this and/or will reveal useful extensions to XChange.

The EU-Rent[5] case study has been chosen for developing (some of the) XChange use cases; it is a case study used by the business rules community to demonstrate their product abilities. EU-Rent is a (fictive) car rental company with branches in different cities and countries. XChange use cases consider Web-based EU-Rent branches distributed over the network. EU-Rent branches offer typical car rental services such as car reservations (rentals made in advance) or 'walk-in' rentals, rentals of cars from different car groups are possible, and customers may return cars to different branches. Business rules used by the EU-Rent branches for constraining or controlling (some of their) processes (actions that might be part of workflows) regard e.g. car reservations, offered discounts, or car assignment. Some concrete examples follow:

(i) If the customer requesting a car rental has no valid driver licence, the request should be denied.

(ii) If the car model requested for reservation is not available, a car of the same group should be assigned.

For flexible changes to business rule and processes, business rules should be separated from the processes, not contained in them; however, different implementation approaches are conceivable. In general, the approach chosen depends on the intended applications; for example, for the EU-Rent case study, business rules could be stored at each branch or just at the central branch(es) coordinating (local) ones. Moreover, different approaches for so-called *decision points* (between steps of a workflow when a decision is to be taken based on the set of employed business rules) are possible. For example, business rules can be "verified" before executing each action (or process) of the workflow, or business rules can be "verified" when events occur during workflow execution.

The EU-Rent case study, as a concrete example where business rules and reactivity on the Web are employed, and the implementation in XChange of (some of) the possible scenarios are ongoing work. Inna Romanenko investigates this issues as her master thesis under the double supervision of Prof. Dr. François Bry and the author. This work has begun recently, but first results are promising. The EU-Rent scenarios can be combined with *Travel organisation* scenarios so as to support the travel organisation task with the possibility of renting cars for the trips; this is one of the possible directions for future work.

[3]Business Rules Group, `http://www.businessrulesgroup.org/brghome.html`

[4]RuleML, `http://www.ruleml.org`

[5]EU-Rent, `http://www.eurobizrules.org/eurentcs/eurent.html`

Part III

Conclusion

CHAPTER
SEVEN

Conclusion

7.1 Contributions

The research topic investigated by this thesis is *reactivity on the Web*. Identifying possible approaches to reactivity on the Web calls for analysing the characteristics of the Web – as environment for applications that exhibit reactive behaviour – so as to suit the reactive technology to them. Turning then to what reactivity means and keeping in mind that the environment is a distributed one, reactive technologies call for: updating data on the Web, exchanging information about events (such as executed updates) between reactive Web systems, and automatically reacting to combinations of such events. Moreover, of crucial importance for the Web (and also for the Semantic Web) is the lightness of technologies' usage (in particular the languages' usage) that should be approachable also by non-programmers.

Following a declarative approach to reactivity on the Web, a novel *reactive language* called *XChange* is proposed. XChange is a high-level programming language that offers means for programming distributed Web applications requiring state changes as reaction to events that have occurred on the Web. It consists of three components: an event query language, a Web query language, and an update language. XChange has an imperative nature, but its components do follow a declarative approach: The event query language of XChange offers means for declaratively specifying classes of events of interest that might require a reaction. The Web query language of XChange is inherited from the declarative Web and Semantic Web query language Xcerpt, i.e. the 'condition part' of XChange reactive rules are Xcerpt queries and Xcerpt (construct-query) rules can be specified in an XChange program for constructing views over Web data. The update language of XChange offers means for declaratively specifying templates for the data before the updates, and templates for the data to be inserted, removed, or replaced. Though, an update language with explicit update operations (expressing e.g. `do an insert`) can not be fully declarative; also, a conjunction of updates to be executed in sequence has an imperative touch.

The research work presented in this thesis (having as materialisation the reactive language XChange) contributes to the research on (Semantic) Web reactivity with the following:

Recognised Requirements for Web Reactivity The development of application scenarios for reactivity on the Web has raised requirements for languages aimed at entailing enhancements of the actual Web with reactive capabilities. To mention some of these requirements: composite event queries and composite event detection, and event-driven transactions are between the first-class citizen of needed language abilities. However, different classes of applications exist that might be useful for upcoming, reactive Web systems; these share with applications like the ones presented in Section 1.2 and Chapter 6 some capabilities but, at the same time, might require new ones or just adapting the presented approach to the considered application domain.

Concept Clarification This thesis offers clear definitions and descriptions of the concepts and notions used by/in XChange for programming reactive applications. Most proposals for reactivity use notions that do not always have an unambiguous meaning; overloading notions (for example, by not differentiating between *event* and *event query*) precludes a clear language semantics and thus, makes the implementation of the language and its usage much more difficult.

Novel and Intuitive View over (Reactive) Web Data XChange introduces a novel view over the Web data by stressing a clear separation between *persistent data* (data of Web resources, such as XML or HTML documents) and *volatile data* (event data communicated between XChange programs on the Web). Based on the differences between these kinds of data, the data metaphor is that of *written text* vs. *speech*. XChange's language design enforces this clear separation and entails new characteristics of event processing on the Web.

Language Design The design of the language XChange has received special attention throughout the whole development process; however, it is not that clear which kind of constructs should necessarily be included into a reactive language developed not only for a single kind of applications, but trying to cover different classes of applications. A tradeoff between the expressive power of the language and the ease of its usage has been looked for in designing XChange. On the other hand, the language XChange aims at acting as a 'referee' language where the pattern-based approach has been investigated and used for specifying reactivity on the Web.

Event Queries with Double Purpose One of the essential traits of XChange event queries is that they have a double purpose: they are aimed for *event detection* and *event data extraction*. Event queries in XChange not only detect atomic and composite events that have occurred on the Web and might require a reaction, they also extract pieces of information from incoming events by means of *variables*. The bindings for the variables occurring in event queries can be subsequently used for raising events or executing updates.

Composite Event Queries Novel in XChange is its ability to detect *composite events* on the Web, i.e. possibly time related combinations of events that have occurred at (same or different) XChange-aware Web sites. This is possible as XChange offers *composite event queries* for specifying interest in classes of events only if they are in particular temporal relationships.

Bounded Event Lifespan Events are not stored forever in memory, just as long as they are needed for answering the event queries registered at a Web site. The amount of time for which data on any received event is in memory – the *event lifespan* – is bounded. This is aided by the fact that event queries are *forward-looking* (event queries do not query events received in the past). By design, the property of bounded event lifespan is enforced.

Declarative Semantics Declarative semantics are not only beneficial to avoid misinterpretations of language constructs by both users and implementors; they also provide a basis for formal proofs of language properties. However, the declarative semantics of other proposals for Web reactive languages is not provided. A model theoretical semantics for XChange event query language has been defined as a ternary relation between an event query, an answer, and the stream of incoming events. The Web query language Xcerpt integrated into XChange provides a model theoretical semantics for the Web queries of reactive rules and for deductive rules. Moreover, this work on declarative semantics is used for the XChange update language, as the effect of an elementary update is the same as of a corresponding Xcerpt goal.

Use Cases Developing use cases for a language aims at introducing, removing, or modifying constructs of the language. It also reveals the strengths and limits of a language. The most substantial contribution of the use cases developed for the language XChange (some of them presented or touched on in this thesis, and some representing ongoing work) is that they bear evidence for the practicability of language constructs.

7.2 Perspectives

The language XChange is an approach to reactivity on the Web; XChange has not saturated the research on reactivity on the Web, instead it aims at acting as a framework for further research work. Some of the perspectives for further work in XChange follow.

7.2.1 Transaction Management on the Web

The work on XChange has recognised the need for transactions (as combinations of possibly complex updates and events to be raised that are to be executed in an all-or-nothing manner) through developed application scenarios and the components a transaction on the Web might have (cf. Section 4.7). XChange proposes a syntax for (event-driven) transactions on the Web by specifying them as 'action part' of XChange transaction rules (cf. Section 4.8.2). To some extent, the execution of actions in an all-or-nothing manner can be implemented by means of XChange reactive rules. However, as the discussion on transactions in Section 4.7.3 has shown, issues (such as the cascading triggering of local and remote actions inside a transaction) need to be investigated and means need to be materialised for *transaction management on the Web*. Database technology applied to Web data (such as XML or RDF) and the Web as environment can play an important role in realising this task. However, as Jennifer Widom already recognised "everything needs to scale to Web proportions (!)."[1] Along this line, work on transactions in database management systems such as [141, 66, 77] might prove very useful.

7.2.2 Generation of XChange Rules

The issue of generating XChange rules based on integrity constraints' specifications is a promising perspective for further work. The management of distributed projects is a good, concrete example where generating XChange reactive rules would be of real help. For example, the research project REWERSE (Reasoning on the Web with Rules and Semantics) has 27 participants each contributing to the project with its own members. At site `http://rewerse.net` information about the project, its participants, and its members can be found. Updating a member's information results in updating multiple, distributed Web documents (e.g. home page data, member page data, working group involvement data). At moment this is done manually implying an amount of time and (surely) delays in gaining data accuracy. Thus, a kind of 'integrity-preserving rules' need to be generated. This issue has been investigated in the context of active database systems, where different possibilities have been explored: (i) syntactic generation of event and condition parts, (ii) syntactic generation of event and condition parts, and declarative specification of action part, and (iii) syntactic generation of event and condition parts, and semantic generation of action part. A detailed discussion on these possibilities accompanied by examples and references can be found in [141], pages 264-270.

7.2.3 Efficiency Issues

This thesis has not considered optimising XChange event query evaluation and XChange update execution; a more efficient evaluation of Web queries is investigated at moment in the Xcerpt project. However, they leave room for research work aiming at a more efficient event query evaluation and update execution on the Web. One direction towards efficient event query evaluation is the *clustering of XChange rules*; the idea is to recognise event queries or parts of them that occur in more than one rule and to cluster rules into rule sets that can be associated with event classes. This method might be useful when the number of reactive rules is big (e.g. when implementing a large number of business rules by means of reactive rules). Clearly, a basis for a more efficient execution of updates on the Web would be provided by an update execution on secondary storage.

[1] Jennifer Widom, Data Management for XML, urlhttp://www-db.stanford.edu/ widom/xml-whitepaper.html

7.2.4 Type System for Semistructured Data

The integration into XChange of a type system for semistructured data with static type checking ability is planned. Type checking of the language XChange means detection prior to execution that event queries and Web queries do not provide results based on the types the queries and the data to be queried have (clearly, if the schema of event data and Web resources' data is known), and executing the specified update terms do not result in updating data because e.g. the corresponding subjacent query terms do not match the data to be modified.

The idea is to extend the type system for the Web query language Xcerpt so as to cope with the constructs of the event query and update languages of XChange. At moment two research efforts towards Xcerpt's type system exist: R^2G^2 (Rooted Regular Graph Grammars) [22], and the type system proposed in [41]; both are regular tree grammar-based approaches for typing Xcerpt.

7.2.5 Visual Rendering of XChange Programs

Visual renderings of programming languages play an important role in easing the use of languages by programmers and novice practitioners. The intended visual counterpart of XChange extends the visual language visXcerpt [23, 26, 27], merely a visual rendering of textual Xcerpt programs. The idea is to keep and extend the clear design principles that have been followed in developing visXcerpt so as to obtain a very easy to use visual reactive language for the Web. Some of the design principles for the visual rendering of XChange elementary updates have been already identified, e.g.

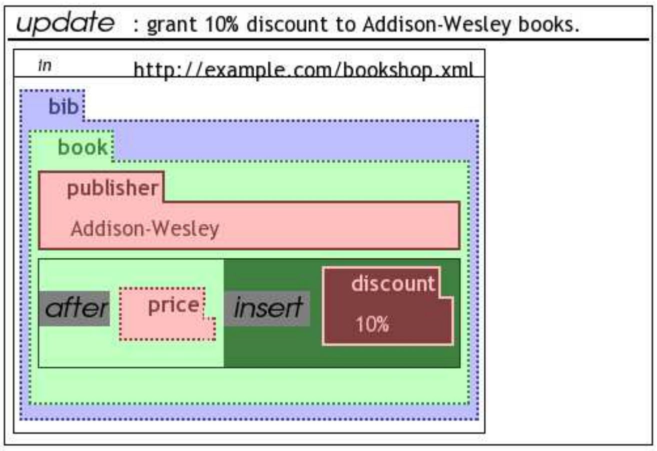
(i) grey coloured boxes labelled with `insert`, `delete`, or `replace by` visualise the simplest update operations,

(ii) insert and delete boxes are visualised on the left, 'before' the (construct or query) pattern, and replace boxes are visualised between the query and construct patterns,

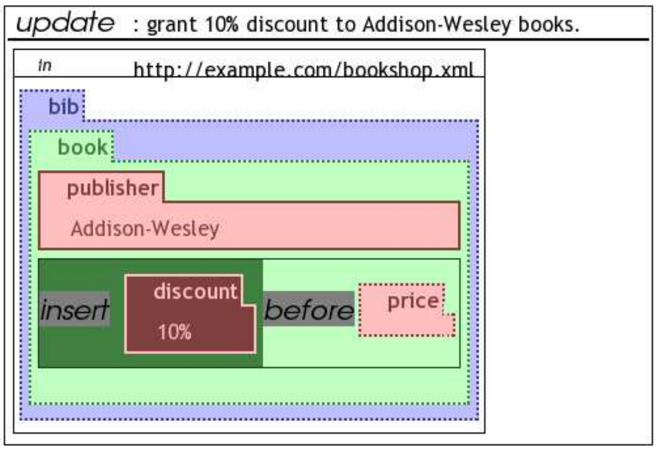
(iii) patterns for the data before the update are visualised using light-coloured boxes, patterns for the data after the update is performed are visualised using corresponding dark-coloured boxes.

The following examples bear evidence for the simplicity of the implied visualisation.

Example 7.1 (Visual Rendering of XChange Insertion After)
The following example visualises an XChange elementary update specifying insertion of discounts of 10 percent for Addison-Wesley books. The `discount` terms are to be inserted after the `price` terms.

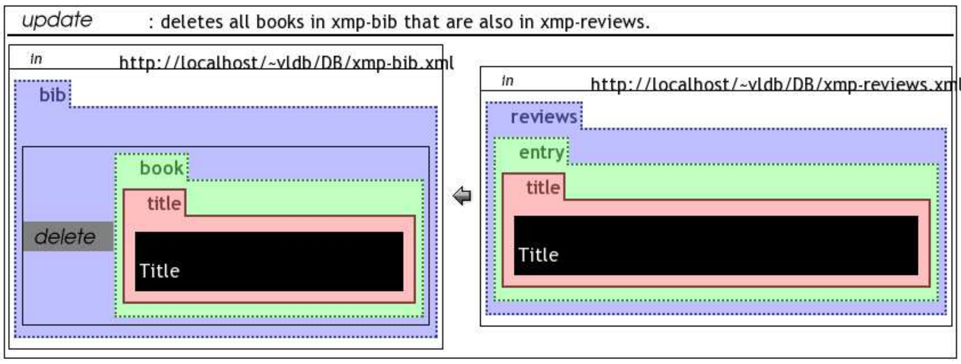

update : grant 10% discount to Addison-Wesley books.
in
http://example.com/bookshop.xml
bib
book
publisher
Addison-Wesley
after
price
insert
discount
10%

Example 7.2 (Visual Rendering of XChange Insertion Before)
The following example is a slight modification of the one given above, where the `discount` terms are to be inserted before the `price` terms.

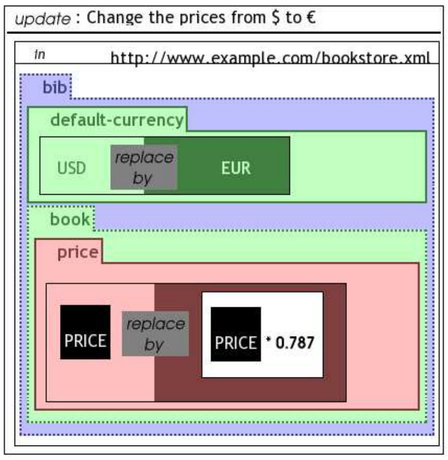

update : grant 10% discount to Addison-Wesley books.
in http://example.com/bookshop.xml
bib
book
publisher
Addison-Wesley
insert
discount
10%
before
price

Examples 7.1 and 7.2 show one of the advantages of a visual rendering of the update language – the structure of the data after the update is performed is very easy to grasp.

Example 7.3 (Visual Rendering of XChange Delete)
The following example visualises an XChange rule having just 'condition' and 'action parts'; the rule is used for deleting all books from `xmp-bib.xml` that are also in `xmp-reviews.xml`. The 'condition part' is given on the right, the 'action part' on the left, and the parts are connected using an arrow.


update : deletes all books in xmp-bib that are also in xmp-reviews.
in http://localhost/~vldb/DB/xmp-bib.xml
bib
delete
book
title
Title
in http://localhost/~vldb/DB/xmp-reviews.xml
reviews
entry
title
Title

Example 7.4 (Visual Rendering of XChange Replace)
The following example visualises an XChange elementary update that is used for changing the prices from Dollar to Euro for all books found at `http://www.example.com/bookstore.xml`.


update : Change the prices from $ to €
in http://www.example.com/bookstore.xml
bib
default-currency
USD
replace by
EUR
book
price
PRICE
replace by
PRICE * 0.787

For developing a visual counterpart to the whole reactive language XChange, analysing the design principles of other existing visual query and transformation languages such as XML-GL [58, 59], GraphLog [71], or VXT [121] might prove useful.

7.2.6 Authentication, Authorisation, and Accounting

XChange in its present stage of development does not offer specific means for security (especially authentication and authorisation). However, such extensions are neither incompatible with the current version of XChange nor precluded in future versions of the language. The protocols of a Grid architecture (such as Globus [79]) would provide with convenient means for such an extension. Extending XChange with accounting functionalities is a promising perspective for future research. Vice versa, XChange could be seen as a (core of a) high-level reactive language for Grids.

7.2.7 Integration with Location and Temporal Reasoning Languages

The integration with languages for specifying and reasoning with specific kind of data is intended. The presented application scenarios assessed the practical need of reasoning with location data (e.g. to look for and book a hotel in a quiet area near to a metro station). This suggests to consider an integration of XChange with MPLL (Multi Paradigm Location Language) [25, 45], a language for specifying and reasoning with different kinds of location data. Integrating XChange with CaTTS (Calendar and Temporal Type System) [51, 44], a static typed calendar and time language that allows for declarative modelling of various calendars (e.g. Gregorian calendar, business calendars, holiday calendars, etc.), would provide the event language with richer temporal specifications.

7.3 Concluding Remarks

This thesis has presented the language XChange, a programming language that aims at filling the gap between the actual, passive Web and the more dynamic, reactive Web. It suits to building applications based on the Web as a large distributed environment by considering local programs through which global behaviour is achieved. XChange's language design enforces the clear separation between persistent (Web resources' data) and volatile data (event data) and entails new characteristics of event processing on the Web.

Developing a programming language with clear paradigms mirrored by its syntax, powerful constructs, and a simple formal semantics is not (necessarily) an easy task; though, it is an interesting research work. This kind of work offers strong motivation all along the developing phase, for the language shapes up making one to proceed with the conviction of attaining a useful technology.

After more than two years of research work on XChange, the language is not yet "mature" enough (cf. Section 7.2) to freeze the work on it with the completion of this thesis. The author ends the writing of this thesis by hoping that XChange will act as a framework for investigating further research work on reactivity on the Web.

Part IV

Appendix

APPENDIX

A

A Prototypical Runtime System

This part of the thesis discusses the proof-of-concept implementation that has been developed for the language XChange. At moment of writing, the XChange prototype does not implement all features of the language; the development of the XChange prototype is ongoing work. This part gives a description of its current status accompanied by suggestions on where and how the implementation is to be changed or extended for fully implementing XChange.

The XChange prototype has been implemented in Haskell [135], a functional programming language. Chosing Haskell has been strongly motivated by the existing Xcerpt[1] prototype implementation, which is implemented in Haskell. Recall that not only Web queries and deductive rules specified in Xcerpt need to be evaluated for executing XChange programs, but also Simulation Unification is employed for evaluating XChange atomic event queries. Thus, the prototype implementation of Xcerpt has been "extended" so as to implement the language XChange.

For space reasons, this part of the thesis does not offer a complete discussion of every aspect of the implementation; it offers a high-level guide to XChange's implementation and a general view over the structure of the source code. It also abstracts away from details on the Xcerpt implementation; more information on Xcerpt's prototype implementation can be found in [125], pages 207-225.

A.1 Overview. Source Code Structure

For executing XChange programs, an implementation of the language XChange needs to provide, besides a parser for the language, components for evaluating the 'event part' and the 'condition part', and executing the 'action part' of XChange rules. These components need to communicate through substitution sets for the variables with at least one defining occurrence in the rule parts. The implementation needs to convey the semantics of the three rule parts, which has been presented in Chapter 5.

Mirorring the three parts of an XChange reactive rule, the main module of the XChange prototype implements an *event handler*, a *condition handler*, and an *action handler*. They are defined as functions and run separately. Communication between the functions implementing the event, condition, and action handlers is realised through *channels*, an extension to Haskell found in Concurrent Haskell [120]. Channels provide a buffered First In, First Out message-pasing communication between the component handlers. The flow of messages between the handlers mirrors the flow of substitution sets between the parts of an XChange reactive rule.

The event handler has the abilities to receive atomic events, evaluate the event queries registered in the system, and release events whose lifespan has expired. Detected answers to the these event queries are communicated to the condition handler through a *condition channel*. Upon successful evaluation of an event query *eq*, the condition handler evaluates the Web query *q* of the rule having *eq* as event query (for determining which parts form a registered reactive rule, each rule gets an identifier at registration); evaluation of Web queries is based on Xcerpt's abilities. The successful evaluation of the Web query *q* is

[1]Xcerpt Project, `http://www.xcerpt.org`

signalled to the action handler by writing the substitution set obtained from the evaluation of *eq* and *q* to the *action channel*. The action handler executes the action of the rule having 'event part' *eq* and 'condition part' *q*. It executes local updates by transforming update terms into Xcerpt goals and evaluate these goals. At moment, the prototype implementation of XChange does not offer support for executing transactions on the Web.

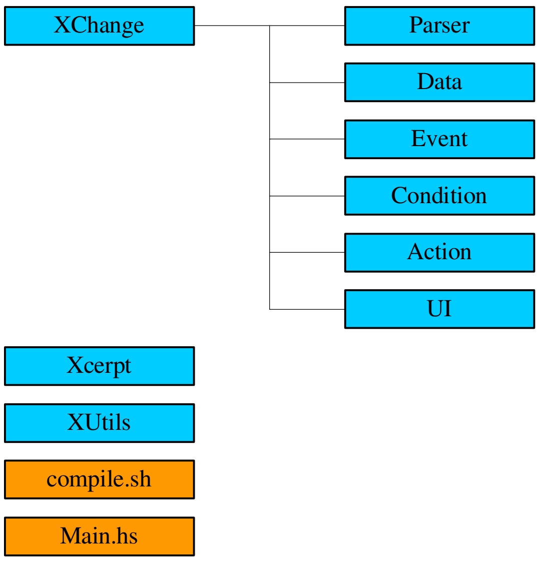


Figure A.1: Overall module and file structure

The structure of the source code is depicted in Figure A.1. By using the hierarchical module mechanism of Haskell, the code is structured in different modules: Module `XChange` implements the language XChange by using module `Xcerpt`, which implements the language Xcerpt. Module `XUtils` implements some date, list, and string utilities (i.e. functions needed in module `XChange`). Files `compile.sh` and `Main.hs` are used for compiling XChange (see Section A.7). Module XChange is made of the following submodules:

XChange.Parser provides lexer and parser modules for parsing XChange programs into the data structures defined in `XChange.Data`.

XChange.Data provides data structures (e.g. data structures into which programs are parsed and channel data structures) and functions on these structures.

XChange.Event provides functions for receiving events, evaluating event queries, and deletion of events.

XChange.Condition provides functions for evaluating Web queries and deductive rules.

XChange.Action provides functions for executing actions (e.g. for transforming update terms into Xcerpt goals and evaluate these goals).

XChange.UI provides functions for the command line and for debugging purposes.

The next sections discuss in more detail the functions and data structures defined in the modules `XChange.Parser`, `XChange.Data`, `XChange.Event`, `XChange.Condition`, and `XChange.Action`. The module `Xcerpt` is described in [125].

A.2 XChange Parser

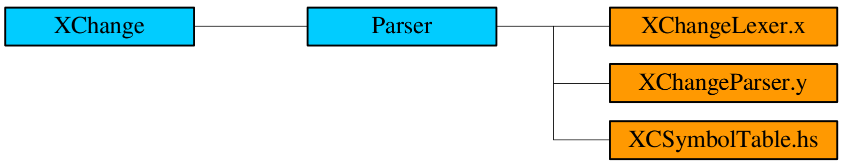


Figure A.2: Module and file structure of XChange.Parser module

Given an XChange program to be executed, two steps are realised for transforming the program into the data structures used by the event, condition, and action handlers: A lexical analysis is performed for transforming the characters of the program into a sequence of tokens. Based on the language grammar, the sequence of tokens is transformed into XChange data structures (defined in module `XChange.Data`).

The module and file structure of the `XChange.Parser` module is presented in Figure A.2. The XChange parser has been implemented by extending the parser for Xcerpt programs. The lexical analysis of XChange programs is done by an XChange lexer built by using the lexer generator Alex [75]. The parser of XChange programs has been built by using the parser generator Happy [107].

Module `XChange.Parser.XChangeLexer` implements the XChange lexer; it defines tokens in terms of regular expressions. The function `lexer` in `XChangeLexer.x` takes a string (an XChange program) and return a list of tokens ([`Token`]). Module `XChange.Parser.XCSymbolTable` defines the function `resolveSymbols` that looks up in a symbol table for correctly analysing XChange programs. The symbol table contains tokens for the language keywords; it should be modified when XChange constructs are modified or extended.

Module `XChange.Parser.XChangeParser` defines the function `parseXCProgram` for parsing the list of tokens resulted from the lexical analysis of a program. The parser generator Happy uses grammar rules in a syntax similar to Backus-Naur Form (BNF); rules are extended for defining the action to be taken when “encountering” given specifications. For example, the next given code specifies an excerpt of the grammar rule defining XChange event query specifications. An event query specification is either a term specification, or a keyword (here `or`, `and`, `andthen`) followed by single or double opening braces, a list of event query specifications, and corresponding closing braces. An event query (instance of type `EvQuery`) is to be returned.

```
PEvQuery :: { EvQuery }
PEvQuery : PTerm                           { EvQTerm $1 }
       | or  '{' PEvQueryL '}'             { EvQOr $3 }
       | and '{' PEvQueryL '}'             { EvQAnd $3 }
       | andthen '[' PEvQueryL ']'         { EvQAndthen $3 False }
       | andthen '[' '[' PEvQueryL ']' ']' { EvQAndthen $4 True }
       ...
```

Clearly, `XChangeParser.y` contains the grammar rules for XChange event queries, Web queries, actions, and rules. Such grammar rules are used for building the data structures introduced in the next section. The function for performing the lexical analysis and building the XChange data structures is also defined in `XChangeParser.y`:

```
parseXChange = parseXCProgram . resolveSymbols . lexer
```

A.3 XChange Data Structures

The module `XChange.Data` defines the data structures on which the event, condition, and action handlers work; it also provides functions over these data structures. The module and file structure of the

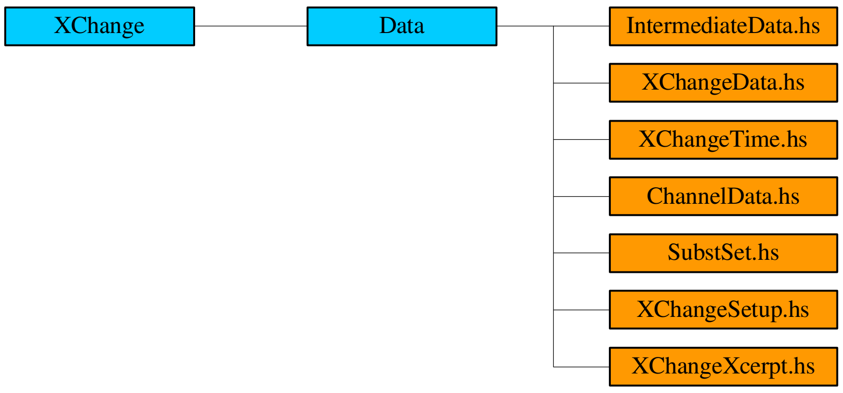


Figure A.3: Module and file structure of XChange.Data module

`XChange.Data` module is given in Figure A.3; the module is made of the following submodules:

XChange.Data.IntermediateData defines the data structures (Haskell data types) whose “instances” are built when parsing XChange programs. Recall the previous example giving an excerpt of the parser’s code; it returns an instance of the type `EvQuery`. The following code gives an excerpt of the definition for the data type `EvQuery` (corresponding to an XChange event query):

```
data EvQuery = EvQTerm { eterm :: Term }
             | EvQOr  { evq_queries :: [EvQuery] }
             | EvQAnd { evq_queries :: [EvQuery] }
             | EvQAndthen { evq_queries :: [EvQuery], evq_partial :: Bool }
             ...
             deriving (Eq,Show)
```

For example, the data type for a temporally ordered conjunction event query is given as a constructor `EvQAndthen` followed by a list of elements of type `EvQuery` and a boolean value expressing whether the event query specification is total or partial. The data type corresponding to an XChange program is defined as:

```
data XCProgram = XCProg [XCRule] deriving Show
```

That is, the XChange parser “transforms” an input XChange program into a list of rules (elements of type `XCRule`) having as constructor `XCProg`.

XChange.Data.XChangeData defines a slight modification of the data types returned by the XChange parser; these types are further used by the event, condition, and action handlers. For example, a rule identifier type is defined here that is to be associated to each rule registered in the system. Also, the time specifications (time points, time intervals, durations) returned as strings by the parser are transformed into XChange time types (e.g. type `XChangeDuration` for a length of time).

XChange.Data.XChangeTime defines the XChange time types (`XChangeTime`, `XChangeDuration`) that are needed for evaluating event queries correctly. The module provides also functions for transforming strings in XChange time types and the relations between time points and durations, respectively (e.g. equality relation, comparison of time points and durations, respectively). For example, the data type `XChangeDuration` is defined as a time difference or an integer having as constructors `XChangeTimeDiff` and `XChangeIntDuration`, respectively:

```
data XChangeDuration
    = XChangeTimeDiff TimeDiff |
      XChangeIntDuration Int
```

and the equality relation on durations:

```
instance Eq XChangeDuration where
    (XChangeTimeDiff td1) == (XChangeTimeDiff td2)
      = (td1 == td2)
    (XChangeIntDuration i1) == (XChangeIntDuration i2)
      = (i1 == i2)
```

The XChange time types are used also for the parameters of XChange event messages, i.e. raising time and reception time.

XChange.Data.ChannelData defines data structures `AtomicEvent` (as a term with a reception time) and `CompositeEvent` (as a list of `AtomicEvent` instances, a beginning and an ending time, and a constraint). Constraints represent the possible variable substitutions; they are used in defining the data structure `Firing`, instances of which are communicated through the channels expressing successful evaluations of parts of an XChange rule (identified by `XCRuleId`). Functions on these data structures are also provided (e.g. show functions for firings output).

XChange.Data.XChangeXcerpt, XChange.Data.SubstSet provides data structures and functions from Xcerpt that are needed for evaluating event queries and Web queries.

XChange.Data.XChangeSetup defines configuration data used for executing XChange programs. The event, condition, and action channels are defined in this module; also functions for the output of e.g. intermediate states or firings, and other useful data (e.g. port number for server) are declared.

A.4 XChange Event Handler

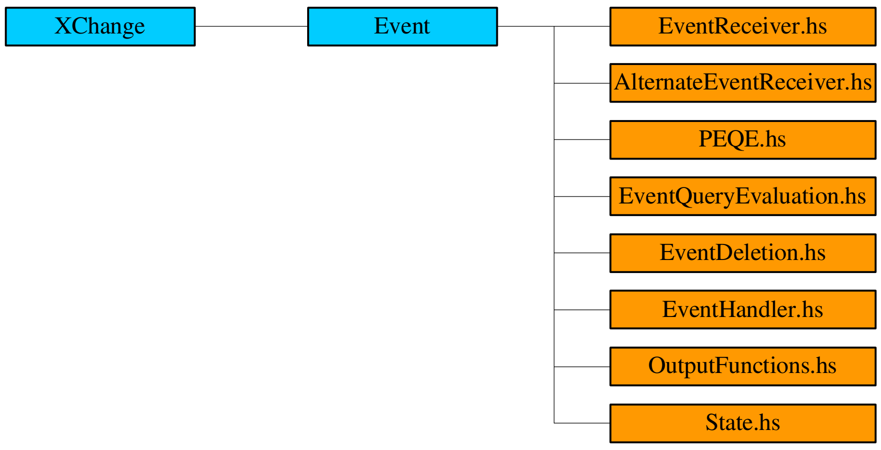


Figure A.4: Module and file structure of XChange.Event module

The XChange event handler receives event messages, evaluates the event queries registered in the system, and deletes events whose lifespan has expired. The module and file structure of the module `XChange.Event` is given in Figure A.4. The module `XChange.Event` consists of the following submodules:

XChange.Event.EventReceiver defines the function `eventReceiver` for receiving event messages from TCP/IP connections using the default port 4711. The received event messages are augmented with reception time and identifier before writing them in the event channel for the event handler:

```
let ae = (AtomicEvent term (XChangeClockTime rcptTime))
writeChan channel ae
```

The prototype uses TCP/IP socket communication; receiving and sending event messages over HTTP is planned.

XChange.Event.AlternateEventReceiver is used for evaluating event queries against the event messages contained in given files; it is useful for debugging purposes.

XChange.Event.PEQE implements the operator trees for XChange event queries; a discussion on the defined data structures and functions can be found in [76], Section 8.3.

XChange.Event.EventQueryEvaluation defines the function `evaluateQuery` that performs the event query evaluation:

```
evaluateQuery :: AtomicEvent -> PartialEventQueryEval ->
                (PartialEventQueryEval, [CompositeEvent])
```

The function takes an atomic event and a partial evaluated event query, it returns an updated partial evaluation and a list of composite events. A more detailed discussion on `evaluateQuery` is given in [76], Section 8.3.

XChange.Event.EventDeletion provides functions for releasing events after their lifespan has expired; the method outlined in Section 5.2.1 is followed.

XChange.Event.EvenHandler defines the main loop of the event handler, a tail-recursive function

```
eventHandlerLoop :: XChangeSetup -> State -> IO()
```

where the state of the event handler is a list of partial evaluations (`PartialEventQueryEval`) for the event queries of the rules registered in the system (identified by `XCRuleId`):

```
newtype State = State [(PartialEventQueryEval, XCRuleId)]
```

XChange.Event.OutputFunctions provides functions for the output of received events, firings, and registered rules.

XChange.Event.State provides the definition of type `State` given above.

A.5 XChange Condition Handler

The XChange condition handler evaluates Web queries and deductive rules when a new firing is signalled from the event handler. The evaluation of Web queries and deductive rules is based on the prototype implementation of Xcerpt.

The main loop of the condition handler (`conditionHandlerLoop`) is implemented as a tail-recursive function so as to get an infinite loop in Haskell. The type of the function is

```
conditionHandlerLoop :: XChangeSetup -> State -> Program -> IO()
```

where the state of the condition handler is a list of Web queries (`Query`) registered in the system and associated with the corresponding rule identifiers (`XCRuleId`):

```
newtype State = State [(Query, XCRuleId)]
```

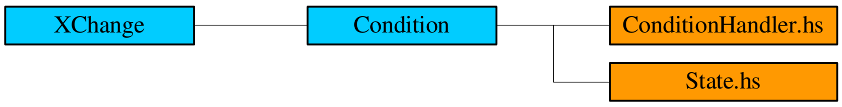


Figure A.5: Module and file structure of XChange.Condition module

and the program (of type `Program`) contains the deductive rules of the XChange program to be executed. The result type of the function is the `IO()`-monad.

The module and file structure of the module `XChange.Condition` implementing the XChange condition handler is depicted in Figure A.5. The submodules of `XChange.Condition` are:

XChange.Condition.ConditionHandler defines the function presented above that implements the condition handler. When a new firing is written in the condition channel, the condition handler looks for the Web query associated with the rule identifier (recall that a firing is made of a constraint and a rule identifier) in its state. The Web query is evaluated against the specified Web resources (if a resource specification is given) or against the set of deductive rules (`xcerptRules`) contained in the XChange program executed.

The evaluation returns either a constraint `False` (expressing unsuccessful evaluation) or a constraint expressing possible bindings for the variables; in the latter case, a new firing is written to the action channel representing the modified constraints (obtained from the event handler and condition handler) associated with the rule identifier.

XChange.Condition.State provides the definition of type `State` given above.

A.6 XChange Action Handler

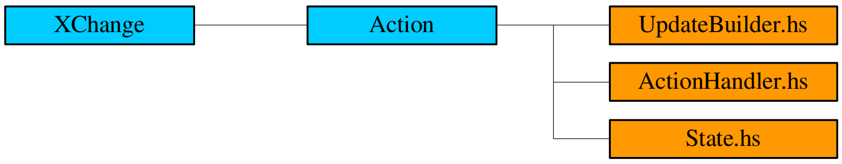


Figure A.6: Module and file structure of XChange.Action module

The XChange action handler executes the actions specified in the ’action part’ of XChange reactive rules. Similar to the functions implementing the event and condition handlers, the main loop of the action handler (`actionHandlerLoop`) is a tail-recursive function:

```
actionHandlerLoop :: XChangeSetup -> State -> IO()
```

where the state of the action handler is a list of tuples of actions registered in the system (`XCAction`) and corresponding rule identifiers (`XCRuleId`):

```
newtype State = State [(XCAction, XCRuleId)]
```

Again, the result type of the function is the `IO()`-monad.

The module and file structure of the `XChange.Action` module is given in Figure A.6. The submodules of `XChange.Action` are:

XChange.Action.UpdateBuilder defines the function `createUpdateGoal` that takes an update term, a list of resources to be modified and returns an Xcerpt goal that is used to construct the data after the update:

```
createUpdateGoal :: Term -> [Resource] -> Rule
```

The function implements the needed rewriting rules for transforming XChange update terms into Xcerpt goals; the rules are given in Appendix B.

XChange.Action.ActionHandler defines the `actionHandlerLoop` function discussed above. Upon reception of a new firing consisting of a constraint and a rule identifier from the condition handler, the action handler looks for the action to be executed (using its state and the rule identifier). The function `executeAction` executes the retrieved action; at moment, it implements the execution of local updates by constructing data after the update.

XChange.Action.State provides the definition of type `State` given above.

At present, the action handler executes XChange local updates as reactions to (atomic or composite) events. The execution of remote updates and raising events are to be implemented in the near future. The implementation of local updates is the most important part of the action handler; it acts as a building block for the execution of remote updates. For executing a remote update u_r specified in an XChange program P, P sends a request to the XChange processor at the Web site whose data is to be modified by u_r. Thus, the desired update u_r is to be executed locally by the XChange processor receiving the update request.

Raising and sending event messages can be easily implemented: The given event term(s) and the constraints (variable substitutions) received through the action channel are used to construct data term(s) to be sent. The construction of data terms can be realised by using the `applySubstitutions` function of Xcerpt (defined in module `Xcerpt.EngineNG.Substitution`):

```
applySubstitutions :: Term -> [Substitution] -> [Term]
```

and the function `getSubstSet` (defined in module `XChange.Data.SubstSet`) for obtaining the set of substitutions from the constraint received through the action channel:

```
getSubstSet :: Constraint -> SubstSet
```

where the type `SubstSet` is defined in module `XChange.Data.SubstSet` as

```
newtype SubstSet = SubstSet [Substitution]
```

The obtained data terms need to be augmented with the event messages' parameters `sender` and `raising-time` by using a function similar to `augmentEventTerm` defined in module `XChange.Event.EventRecei-ver`. The sending of the constructed event messages can be implemented similarly to the reception of event messages (see `eventReceiverLoop` in module `XChange.Event.EventReceiver`).

A.7 Building and Running XChange

The source code of the XChange prototypical implementation is available at `http://www.pms.ifi.lmu.de /mitarbeiter/patranjan/`. For building and running XChange, one needs to compile XChange with the Glasgow Haskell Compiler[2] (GHC); the version of GHC used in compiling the XChange source code is GHC 6.2.2. The shell script `compile.sh` calls GHC on the given file. In a Unix shell, one needs to do

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

Now, one can execute XChange programs. The command line syntax for running an XChange program is:

[2]The Glasgow Haskell Compiler, `http://www.haskell.org/ghc/`

```
> xchange [Options] Program [Event Messages Files]
```

where

Options are the supported command line options; they are prefixed by − and provided in a short and a long form (as is common on Unix systems). The options provided by `xchange` are given in the following:

```
Short form  Long form                    Description

-r[FILE]  --receivedEventOutput[=FILE]  write output of received events to FILE
-i[FILE]  --intermedEventOutput[=FILE]  write output of intermediate state to
                                        FILE
-c[FILE]  --cleanedStateOutput[=FILE]   write output of cleaned state to FILE
-f[FILE]  --firingsOutput[=FILE]        write output of firings to FILE
-d[FILE]  --debugOutput[=FILE]          write debug output to FILE
-p[PORT]  --port[=PORT]                 set server port (default: 4711)
```

Program gives the XChange program to be executed (e.g. `./test/test5.xchange`); the current XChange prototype runs XChange programs written using the term syntax of the language. An XChange parser for an XML-based syntax of the language is to be developed.

Event Messages Files give the files from which the event messages are to be used for evaluating the given XChange program; this option is provided for debugging purposes.

The prototype implementation of XChange is not the result of the work of a single person. The implementation of Xcerpt, which is the query language integrated into XChange, is the outcome of Dr. Sebastian Schaffert's efforts with contribution of a couple of graduate students. The evaluation of XChange event queries (for atomic and composite event detection) has been developed as part of the master's thesis of Michael Eckert, work supervised by Prof. Dr. François Bry and the author. The integration of the Xcerpt and XChange implementations as well as other implementation tasks (the transformation of the rewriting rules for update terms from an "informal" description into Haskell code) have been done in collaboration with Oliver Friedmann, a student assistant in the XChange project.

APPENDIX

B

Updates through Construction: Rewriting Rules

This part of the thesis gives rewriting rules for transforming an XChange elementary update into a corresponding Xcerpt goal, i.e. a goal that constructs the data after the update. This represents the approach taken in XChange for executing elementary updates; the main challenges and ideas of the approach have been presented in Section 5.2.3.

Given an XChange elementary update u, the following code (implementing the rewriting rules) constructs a corresponding Xcerpt goal G of the form $ConstructTerm \leftarrow_g QueryTerm$. The structure of the subjacent query term and the update operations of u are taken into account. The resources of u (i.e. persistent data to be modified) are just 'forwarded' to the query and construct part of the Xcerpt goal.

The following code is found in module XChange.Action.UpdateBuilder (see Appendix A.1 for the module and file structure of the XChange prototype implementation and Appendix A.6 for the module and file structure of the action handler). *Note* that `--` precedes comments in the following function definitions.

```
module XChange.Action.UpdateBuilder (createUpdateGoal) where

-- System-related imports
import IO
import System
import Control.Concurrent.Chan

-- Import data structures and functions over lists
import XUtils.ListUtils
import XChange.Data.XChangeXcerpt
import Xcerpt.Data.Program


-- Creates an Xcerpt goal (of type Rule) from an update
-- term (type Term) and resources (type Resource)
createUpdateGoal :: Term -> [Resource] -> Rule
createUpdateGoal t r =
        let
          (term, query) = createUpdateGoal' t getBaseName
        in
          Goal {output = r,
                rhead = maybeTermToTerm term,
                rbody = termToQuery (maybeTermToTerm query) r}


-- An Xcerpt goal consists of head (type Term) and body (type Term)
type UpdGoal = (Maybe Term, Maybe Term)
```

```
-- Creates an Xcerpt goal given a term (type Term) and a
-- base name (type String) for the fresh variables
createUpdateGoal' :: Term -> String -> UpdGoal


-- The following function is applied to every child of an
-- update term and 'concatenate' the results
createUpdateGoal' e@Elem {children = oldChildren, total = t} base =
  let
     -- Variable basename for children
     childrenBase = getVarName base 0
     -- Variable name
     freshVar = getVarName base 1
     -- Position var name
     posVar = getVarName base 2
     -- See Xcerpt, Comparison.hs
     cmpRoutine = ("compareIntTerm", compareIntTerm)
     -- Generates an all optional var Fresh order by position
     -- for the goal's head
     headVar = Optional SortAll { variables = [posVar], cmp = cmpRoutine,
                                  template = [ (Var freshVar)]}
     -- Generates an position var Fresh1 optional var Fresh2
     -- for the goal's body
     bodyVar = Optional TPos {pos = (Var posVar), content =  (Var freshVar)}
     -- Apply createUpdateGoal' on every child and return a
     -- couple of maybe Term lists
     newChildren = unzip (mapIdx (\a i -> createUpdateGoal' a (sb i)) oldChildren 0)
                     where
                     sb i = getSubBaseName childrenBase i
     -- Children of the goal's head
     headChildren = concatListCond (filterMaybe (fst newChildren)) [headVar] (not t)
     -- Children of the goal's body
     bodyChildren = concatListCond (filterMaybe (snd newChildren)) [bodyVar] (not t)
  in
    -- The goal's head is total and ordered
    (Just e{ordered = True, total = True, children = headChildren},
    Just e{children = bodyChildren})


-- For transforming an insert operation: the construct
-- term occurs in the goal's head and nothing in its body
createUpdateGoal' (Insert term) _ = (Just term, Nothing)


-- For transforming a delete operation: the query term
-- occurs in the goal's body and nothing in its head
createUpdateGoal' (Delete term) _ = (Nothing, Just term)


-- For transforming a replace operation: the construct term
-- occurs in the goal's head and the query term in its body
createUpdateGoal' (Replace a b) _ = (Just b, Just a)


-- Variable need to occur in both parts (head and body) of a goal
createUpdateGoal' (Var s) _ = (Just (Var s), Just (Var s))
```

```
-- Rest remains unchanged
createUpdateGoal' t _ = (Just t, Just t)


-----------------------------------
-- Helper functions are defined next
-----------------------------------

-- Compares two integer terms
compareIntTerm :: Term -> Term -> Ordering
compareIntTerm (TInt a) (TInt b) = compare a b


-- Returns a dummy term given nothing and the associated term otherwise
maybeTermToTerm :: Maybe Term -> Term
maybeTermToTerm (Just t) = t
maybeTermToTerm Nothing = TOr [] []


-- Creates a dummy framework around term
termToQuery :: Term -> [Resource] -> Query
termToQuery t r = QTerm {resources = r, term = t}


-- Creates a variable name
-- getVarName "Test" 5 = "Test5"
getVarName :: String -> Int -> String
getVarName s i = s ++ show i


-- Creates a sub variable name
-- getSubBaseName "Test5" 3 = "Test5_3"
getSubBaseName :: String -> Int -> String
getSubBaseName s i = s ++ "_" ++ (show i)


-- Creates a base var name
getBaseName :: String
getBaseName = "Fresh_"
```

Functions defined in module `XUtils.ListUtils` are used for defining the function `createUpdateGoal`; they are given next.

```
module XUtils.ListUtils where

...

-- Maps a function on array by adding index argument
mapIdx :: (a -> Int -> b) -> [a] -> Int -> [b]
mapIdx f (x:xs) i = (f x i) : (mapIdx f xs (i + 1))
mapIdx _ [] _ = []

-- Returns all Just elements
filterMaybe :: [Maybe a] -> [a]
filterMaybe l = [x | Just x <- l]

-- Concats two lists on codition
concatListCond :: [a] -> [a] -> Bool -> [a]
concatListCond x y True = x ++ y
concatListCond x y False = x

...
```

The given Haskell code implements rewriting rules for transforming XChange elementary updates into corresponding Xcerpt goals. These rules have been implemented for proof-of-concept purposes; they cover a representative "class" of XChange update terms. Ongoing work concerns testing the implemented rules to determine to which extent all possible XChange update patterns are covered and to reveal details that have been possibly neglected.

LIST OF EXAMPLES

INDEX

BIBLIOGRAPHY

[1] Tamino XML Server. website. `http://www.softwareag.com/tamino/`.

[2] Working Draft: Database Language SQL (SQL3). ISO/IEC DBL MAD-007, June 1996. `http://www.inf.fu-berlin.de/lehre/SS94/einfdb/SQL3/sqlindex.html`.

[3] World Wide Web Consortium (W3C). website, 2002. `http://www.w3.org/`.

[4] JTC 1/SC 34. *Standard Generalized Markup Language (SGML).* International Organization for Standardization (ISO), 1986. ISO 8879:1986.

[5] Serge Abiteboul, Bernd Amann, Sophie Cluet, Adi Eyal, Laurent Mignet, and Tova Milo. Active Views for Electronic Commerce. In *Proc. of 25th Int. Conf. on Very Large Databases, Edinburgh, Scotland*, 1999.

[6] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[7] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel Query Language for Semistructured Data. In *Proc. Int. Journal on Digital Libraries*, April 1997.

[8] Asaf Adi and Opher Etzion. Amit – the Situation Manager. In *Very Large Data Bases Journal*, volume 13, pages 177–203, 2004.

[9] Rakesh Agrawal and Narain H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. pages 36–45, 1989.

[10] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proc. 2nd International Workshop on the Semantic Web (SemWeb 2001) at WWW'01*, Hong Kong, May 2001.

[11] José Júlio Alferes, Ricardo Amador, and Wolfgang May. A General Language for Evolution and Reactivity in the Semantic Web. In *Proceedings of Third Workshop on Principles and Practice of Semantic Web Reasoning, Dagstuhl, Germany*. INRIA, September 2005.

[12] José Júlio Alferes, Wolfgang May, and François Bry. Towards Generic Query, Update, and Event Languages for the Semantic Web. In *Workshop on Principles and Practice of Semantic Web Reasoning*. Springer, 2004.

[13] James F. Allen. Maintaining Knowledge about Temporal Intervals. In *Communications of the ACM*, volume 26, pages 832–843, 1983.

[14] James F. Allen. Time and Time Again: The Many Ways to Represent Time. In *International Journal of Intelligent Systems*, volume 6(4), pages 341–355, July 1991.

[15] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer.* The MIT Press, 2004.

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

[17] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Flavours of XChange, a Rule-Based Reactive Language for the (Semantic) Web. 2005.

[18] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Event Queries in XChange. 2005.

[19] James Bailey, François Bry, and Paula-Lavinia Pătrânjan. Composite Event Queries for Reactivity on the Web. In *Proc. of 14th Int. World Wide Web Conference*, Chiba, Japan, May 2005. ACM.

[20] James Bailey and Szabolcs Mikulás. Expressiveness Issues and Decision Problems for Active Database Event Queries. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 68–82, London, UK, 2001. Springer-Verlag.

[21] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action Language for XML. In *Int. World Wide Web Conf.*, Honolulu, Hawaii, USA, May 2002.

[22] Sacha Berger and François Bry. Towards Static Type Checking of Web Query Language. In *Proceedings of 17. Workshop über Grundlagen von Datenbanken, Wörlitz, Germany*. GI, May 2005.

[23] Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web. In *Proceedings of 3rd International Semantic Web Conference, Hiroshima, Japan*, 2004.

[24] Sacha Berger, François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data Retrieval and Reactivity on the (Semantic) Web: A Deductive Approach. In *Collaboration Workshop for the Future Semantic Web, at 2nd European Semantic Web Conference (ESWC 2005)*, May 2005.

[25] Sacha Berger, François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, Paula-Lavinia Pătrânjan, Sebastian Schaffert, Uta Schwertel, and Stephanie Spranger. Reasoning on the Web: Language Prototypes and Perspectives. In *Proc. of European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology*, pages 157–164, London, United Kingdom, November 2004. The Institution of Electrical Engineers, IEE.

[26] Sacha Berger, François Bry, and Sebastian Schaffert. A Visual Language for Web Querying and Reasoning. In *Proc. of Workshop on Principles and Practice of Semantic Web Reasoning, Mumbai, India*, volume 2901 of *LNCS*, 2003.

[27] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Int. Conf. on Very Large Databases (VLDB)*, 2003.

[28] Brian Berliner. CVS II: Parallelizing Software Development. In *Proc. of the Winter 1990 USENIX Conference*, Washington, DC, USA, January 1990.

[29] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite Events for XML. In *13th Int. Conf. on World Wide Web*. ACM, 2004.

[30] Tim Berners-Lee. A Strawman Unstriped Syntax for RDF in XML (online), May 1999.

[31] Tim Berners-Lee. *Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor*. Orion Publishing Group, London, UK, 1999.

[32] Tim Berners-Lee, James Handler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[33] Oliver Bolzer, François Bry, Tim Furche, Sebastian Kraus, and Sebastian Schaffert. Development of Use Cases, Part I. 2005. Deliverable.

[34] Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *18th Int. Conf. on Data Engineering (ICDE2002)*, San Jose, California, February 26 - March 01 2002.

[35] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active Rules for XML: A New Paradigm for E-Services. In *First Workshop on Technologies for E-Services, colocated with VLDB2000*, September 2000.

[36] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *World Wide Web*, pages 633–641, 2001.

[37] Anthony J. Bonner and Michael Kifer. A Logic for Programming Database Transactions. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*. Kluwer Academic Publishers, 1998.

[38] Marco Brambilla, Stefano Ceri, Sara Comai, and Christina Tziviskou. Exception Handling in Workflow-Driven Web Applications. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 170–179, New York, NY, USA, 2005. ACM Press.

[39] 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/`.

[40] Jeen Broekstra, Michel Klein, Stefan Decker, Dieter Fensel, and Ian Horrocks. Adding Formal Semantics to the Web: Building on Top of RDF Schema, 2000.

[41] François Bry, Wlodzimierz Drabent, and Jan Maluszynski. On Subtyping of Tree-structured Data A Polynomial Approach. In *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning, St. Malo, France (6th–10th September 2004)*. REWERSE, Springer-Verlag, 2004.

[42] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems (IJSWIS)*, 1(2), 2005.

[43] François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In *Workshop on Principles and Practice of Semantic Web Reasoning*, volume 3208, pages 34–49. Springer, 2004.

[44] François Bry, Jutta Haußer, Frank-André Rieß, and Stephanie Spranger. Cultural Calendars for Programming and Querying. In *Proceedings of 1st Forum on the Promotion of European and Japanese Culture in Cyber-Societies and Virtual Reality, Laval, France*, April 2005.

[45] François Bry, Bernhard Lorenz, and Stephanie Spranger. Calendars and Topologies as Types - A Programming Language Approach to Modelling Mobile Applications. In *Proceedings of 9th International Conference on Knowledge-Based Intelligent Information and Engineering System, Melbourne, Australia*. Springer-Verlag, September 2005.

[46] François Bry and Massimo Marchiori. Ten Theses on Logic Languages for the Semantic Web. In *Proc. of W3C Workshop on Rule Languages for Interoperability*, Washington D.C., USA, April 2005. W3C.

[47] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th Annual ACM Symposium on Applied Computing (SAC'2005)*, volume 2, pages 1645–1649. ACM Press, March 2005.

[48] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web Paradigms and Applications of the Language XChange. 2005.

[49] François Bry, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Poster Presentation: Xcerpt and XChange - Logic Programming Languages for Querying and Evolution on the Web. In *Proc. of 19th Int. Conf. on Logic Programming*, volume 3132 of *LNCS*, pages 450–451, St. Malo, France, September 2004. Springer.

[50] François Bry, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Xcerpt and XChange: Deductive Languages for Data Retrieval and Evolution on the Web. In *Proc. of Workshop on Semantic Web Services and Dynamic Networks*, volume 51 of *LNI*, pages 562–568, Ulm, Germany, September 2004. GI.

[51] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar Types and Constraints for Web Applications. In *Proc. of 14th Int. World Wide Web Conference*, Chiba, Japan, 2005. ACM.

[52] François Bry and Peer Kröger. A Computational Biology Database Digest: Data, Data Analysis, and Data Management. *Distributed and Parallel Databases*, 13(1):7–42, 2002.

[53] François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, June 2002. (invited article).

[54] François Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Proc. 2nd Int. Workshop "Web and Databases"*, LNCS 2593, Erfurt, Germany, October 2002. Springer-Verlag.

[55] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. Int. Conf. on Logic Programming (ICLP)*, LNCS 2401. Springer-Verlag, 2002.

[56] Martin Bryan. *SGML and HTML Explained*. Addison Wesley, 1997.

[57] Alejandro P. Buchmann, Juergen Zimmermann, José A. Blakeley, and David L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. of 11th Int. Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

[58] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *WWW '99: Proceedings of the Eighth International Conference on World Wide Web*, pages 1171–1187, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[59] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, and Letizia Tanca. Complex Queries in XML-GL. In *SAC '00: Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 888–893, New York, NY, USA, 2000. ACM Press.

[60] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Active Rule Management in Chimera. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 151–176. Morgan Kaufmann, 1996.

[61] Stefano Ceri and Rainer Manthey. Chimera: A Model and Language for Active DOOD Systems. In *East/West Database Workshop*, pages 3–16, 1994.

[62] Sharma Chakravarthy, Jyoti Jacob, Naveen Pandrangi, and Anoop Sanka. WebVigiL: An Approach to Just-In-Time Information Propagation in Large Network-Centric Environments. In *Proc. of 2nd Int. Workshop on Web Dynamics in Conjunction with Eleventh Int. World Wide Web Conference*, Honolulu, Hawaii, 2002.

[63] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. of 20th International Conference on Very Large Data Bases*, pages 606–617, Santiago de Chile, Chile, September 1994. Morgan Kaufmann.

[64] Sharma Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[65] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[66] Panos Kypros Chrysanthis and Krithi Ramamritham. *Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, September 1996.

[67] James Clark. Comparison of SGML and XML. Technical report, World Wide Web Consortium (W3C), 1997.

[68] James Clark and Makoto Murata. *RELAX NG Specification*. `http://relaxng.org/`, 2001. ISO/IEC 19757-2:2003.

[69] Sara Comai and Letizia Tanca. Termination and Confluence by Rule Prioritization. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):257–270, 2003.

[70] Stefan Conrad. A Logic Primer. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*. Kluwer Academic Publishers, 1998.

[71] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *PODS '90: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416, New York, NY, USA, 1990. ACM Press.

[72] DAML Project, http://www.daml.org/language/. *The Ontology Language DAML+OIL*, 2001.

[73] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. In *Eighth Int. World Wide Web Conference*, 1999.

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

[75] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*. `http://www.haskell.org/alex/`.

[76] Michael Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Institute for Informatics, University of Munich, Germany, 2005.

[77] Ahmed K. Elmargarmid. *Database Transaction Models for Advanced Applications*. Data Management Systems. Morgan Kaufmann, April 1992.

[78] Tim Furche et. al. Survey over Existing Query and Transformation Languages. Technical report, EU FP6 Project Reasoning on the Web with Rules and Semantics (REWERSE), 2004.

[79] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. In *Int. Journal of Supercomputer Applications*, 2001.

[80] Piero Fraternali and Letizia Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems (TODS)*, 20(4):414–471, 1995.

[81] Norbert E. Fuchs and Uta Schwertel. Reasoning in Attempto Controlled English. In *Workshop on Principles and Practice of Semantic Web Reasoning*, Lecture Notes in Computer Science, Mumbai, India, 2003. Springer.

[82] Stella Gatziu and Klaus R. Dittrich. SAMOS: An Active Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):23–26, December 1992.

[83] Stella Gatziu and Klaus R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *RIDE-ADS*, pages 2–9, 1994.

[84] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating Active Concepts into an Object-Oriented database System. In *Workshop on Database Programming Languages*, pages 399–415, 1991.

[85] Narain Gehani, H. Jagadish, and Oded Shmueli. Compose: A System for Composite Event Specification and Detection, 1994.

[86] Narain H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. of 17th Conference on Very Large Data Bases*, Los Altos, Barcelona, 1991. Morgan Kaufmann.

[87] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[88] Charles F. Goldfarb and Paul Prescod. *The XML Handbook (Second Edition)*. Prentice Hall, 2000.

[89] IETF. *Internationalized Resource Identifiers (IRIs)*, October 2003. Draft 5, `http://www.w3.org/International/iri-edit/draft-duerst-iri-05.txt`.

[90] Infozone, http://www.infozone-group.org. *Infozone Group*, 2002.

[91] Ulrike Jaeger and Johann Christoph Freytag. An Annotated Bibliography on Active Databases. *SIGMOD Record*, 24(1):58–69, 1995.

[92] Rick Jelliffe. *The Schematron Assertion Language 1.5*. Academia Sinica Computing Centre, 2002. `http://xml.ascc.net/resource/schematron/`.

[93] Hao Jin and Curtis Dyreson. Sanitizing using Metadata in MetaXQuery. In *20th Annual ACM Symposium on Applied Computing (SAC'2005)*. ACM Press, 2005.

[94] Michiko Kasuya. Teaching Features of the Stream of Speech in Japanese Classrooms. Technical report, Open Learning Program, University of Birmingham, 1999.

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

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

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

[98] Sebastian Kraus. Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2004.

[99] Tim Berners Lee, Roy Fielding, and Henrik Frystyk. Hypertext Transfer Protocol (HTTP) 1.0. Internet Informational RFC 1945, 1996.

[100] Tim Berners Lee, Roy Fielding, Henrik Frystyk, Jim Gettys, and Jeffrey C. Mogul. Hypertext Transfer Protocol (HTTP) 1.1. Internet Informational RFC 2068, 1997.

[101] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language. Master's thesis, Technische Universität Darmstadt, Germany, August 2001.

[102] Mark Levene and Alexandra Poulovassilis, editors. *Web Dynamics - Adapting to Change in Content, Size, Topology and Use*. Springer, 2004.

[103] Mengchi Liu. A Logical Foundation for XML. In *Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering (CAISE'02)*, Toronto, Canada, 2002.

[104] Mengchi Liu and Tok Wang Ling. Towards Declarative XML Querying. In *Proc. of the 3rd Int. Conf. on Web Information System Engineering (WISE 2002)*, Singapore, 2002.

[105] Mengchi Liu, Li Lu, and Guoren Wang. A Declarative XML-RL Update Language. In *Proc. Int. Conf. on Conceptual Modeling (ER 2003)*, LNCS 2813. Springer-Verlag, 2003.

[106] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

[107] Simon Marlow and Andy Gill. *Happy User Guide*. `http://www.haskell.org/happy/`.

[108] Wolfgang May. *A Logic-Based Approach to XML Data Integration*. Habilitationsschrift, 2001.

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

[110] Dennis McCarthy and Umeshwar Dayal. The Architecture of an Active Database Management System. In *Proc. of the 1989 ACM SIGMOD Int. Conference on Management of Data*, pages 215–224, New York, NY, USA, 1989. ACM Press.

[111] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a Database Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, 1997.

[112] Peter Naur. Revised Report an the Algorithmic Language ALGOL60. In *Communications of the ACM, Vol.3, No.5*, May 1960.

[113] Orbeon, http://www.orbeon.com/oxf/doc/processors-xupdate/. *XUpdate Processor*.

[114] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange across Heterogeneous Information Sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[115] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Workshop on Semantic Web and Databases*, Berlin, September 2003.

[116] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rules on RDF Metadata in P2P Environments. In *Proc. 2nd Workshop on Metadata Management in Grid and P2P Systems (MMGPS): Models, Services and Architectures*, London, UK, December 2004.

[117] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proc. 3rd Hellenic Data Management Symposium (HDMS'04)*, Athens, Greece, June 2004.

[118] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proc. 3rd Web Dynamics Workshop at WWW'04*, New York, US, May 2004.

[119] Norman W. Paton. *Active Rules in Database Systems*. Springer, 1999.

[120] 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.

[121] Emmanuel Pietriga, Jean-Yves Vion-Dury, and Vincent Quint. VXT: A Visual Approach to XML Transformations. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[122] Jon Postel and Joyce Reynolds. File Transfer Protocol (FTP). Internet Informational RFC 959, Network Working Group, 1985.

[123] Mathieu Roger, Ana Simonet, and Michel Simonet. Toward Updates in Description Logics. In *Proc. Int. Workshop on Description Logics (DL2002)*. Ian Horrocks and Sergio Tessaris, editors, 2002.

[124] Arnon Rosenthal, Upen S. Chakravarthy, Babara T. Blaustein, and José Blakely. Situation Monitoring for Active Databases. In *Proc. of the 15th Int. Conference on Very Large Data Bases*, pages 455–464, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[125] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation, University of Munich, Germany, December 2004.

[126] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Int. Conf. Extreme Markup Languages*, Montreal, Quebec, Canada, 2004.

[127] Manfred Schmidt-Schauß and Gert Smolka. *Artificial Intelligence. Attributive Concept Descriptions with Complements*. Elsevier Science Publ. Ltd, 1991.

[128] Scarlet Schwiderski. *Monitoring the Behaviour of Distributed Systems*. Dissertation, University of Cambridge, United Kingdom, April 1996.

[129] *Sesame: A Generic Architecture for Storing and Querying RDF*, LNCS, Sardinia, Italy, 2002. Springer.

[130] Thomas Sindt. Formal Operations for Ontology Evolution. In *Proc. Int. Conf. on Emerging Technologies (ICET'03)*, Minneapolis, Minnesota (USA), August 2003.

[131] Adam Souzis. RxML 1.0 Specification (online), 2004.

[132] Stanford University, http://www-db.stanford.edu/lore/. *The Lore Project*, 1995-2000.

[133] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *Proc. ACM SIGMOD 2001*, Santa Barbara, California, USA, May 2001.

[134] Jean Thierry-Mieg and Richard Durbin. Syntactic Definitions for the ACeDB Data Base Manager. Technical report, MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, 1992.

[135] Simon Thompson. *Haskell: The Art of Functional Programming*. Addison-Wesley, second edition, 1999.

[136] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.

[137] University of Freiburg, http://www.informatik.uni-freiburg.de/ dbis/florid/. *The Florid System*.

[138] University of Trier, http://www.informatik.uni-trier.de/ ley/db/. *DBLP Bibliography*.

[139] Jennifer Widom. The Starburst Rule System: Language Design, Implementation, and Applications. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):15–18, 1992.

[140] Jennifer Widom. The Starburst Active Database Rule System. *Knowledge and Data Engineering*, 8(4):583–595, 1996.

[141] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[142] Erik Wilde. *Wilde's WWW: Technical Foundations of the World Wide Web*. Springer, 1999.

[143] Niklaus Wirth. *Extended Backus-Naur Form (EBNF)*, 1996. ISO/IEC 14977:1996(E).

[144] Wolfgang May. *The LoPiX System.* University of Goettingen, http://www.dbis.informatik.uni-goettingen.de/lopix/.

[145] World Wide Web Consortium (W3C). *Web Naming and Addressing: URIs, URLs, ...* Website, `http://www.w3.org/Addressing/`.

[146] World Wide Web Consortium (W3C). *Cascading Style Sheets (CSS)*, December 1996. W3C Recommendation, `http://www.w3.org/Style/CSS/`.

[147] World Wide Web Consortium (W3C). *HTML 4.01: The HyperText Markup Language*, 1999. W3C Recommendation, `http://www.w3.org/TR/html401/`.

[148] World Wide Web Consortium (W3C), http://www.w3.org/RDF/. *Resource Description Framework (RDF)*, 1999.

[149] World Wide Web Consortium (W3C), http://www.w3.org/TR/xpath. *XML Path Language (XPath)*, 1999.

[150] World Wide Web Consortium (W3C), http://www.w3.org/TR/xslt/. *XSL Transformations (XSLT)*, 1999.

[151] World Wide Web Consortium (W3C), http://www.w3.org/TR/DOM-Level-2-Events/. *Document Object Model (DOM) Level 2 Events Specification*, 2000.

[152] World Wide Web Consortium (W3C), http://www.w3.org/TR/soap. *Simple Object Access Protocol (SOAP) 1.1*, 2000.

[153] World Wide Web Consortium (W3C). *XHTML 1.0: The Extensible HyperText Markup Language*, 2000. W3C Recommendation, `http://www.w3.org/TR/xhtml1/`.

[154] World Wide Web Consortium (W3C), http://www.w3.org/TR/xsl/. *Extensible Stylesheet Language (XSL)*, 2001.

[155] World Wide Web Consortium (W3C). *XML Linking Language (XLink)*, June 2001. W3C Recommendation, `http://www.w3.org/TR/xlink/`.

[156] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer*, 2001. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`.

[157] World Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes*, 2001. W3C Recommendation, `http://www.w3.org/TR/xmlschema-2/`.

[158] World Wide Web Consortium (W3C), http://www.w3.org/TR/xquery/. *XQuery: A Query Language for XML*, February 2001.

[159] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.1*, February 2004. W3C Recommendation, `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[160] World Wide Web Consortium (W3C). *Namespaces in XML 1.1*, February 2004. W3C Recommendation, `http://www.w3.org/TR/2004/REC-xml-names11-20040204/`.

[161] World Wide Web Consortium (W3C), http://www.w3.org/TR/owl-features/. *OWL Web Ontology Language*, 2004.

[162] World Wide Web Consortium (W3C). *RDF Vocabulary Description Language*, February 2004. W3C Recommendation, `http://www.w3.org/TR/rdf-schema/`.

[163] World Wide Web Consortium (W3C). *RDF/XML Syntax Specification (Revised)*, February 2004. W3C Recommendation, `http://www.w3.org/TR/rdf-syntax-grammar/`.

[164] Detlef Zimmer and Rainer Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proc. of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.

[165] Juergen Zimmermann and Alejandro P. Buchmann. REACH. In *Active Rules in Database Systems*, pages 263–277. 1999.

About the Author

Ms. Paula-Lavinia Pătrânjan, born on June 27, 1978 in Cluj-Napoca, Romania, received her B.Sc. in Computer Science in 2001 at the Babeş-Bolyai University, Cluj-Napoca, Romania. One year later, at the same university, she received her M.Sc. in Intelligent Systems.

In Autumn 2002, she has obtained a grant of the German Foundation for Research (Deutsche Forschungsgemeinschaft, DFG) in the framework of the Ph.D. programme “Logic in Computer Science”, a joint programme between the University of Munich (Ludwig-Maximilians-Universität München) and the Institute of Technology Munich (Technische Universität München).

The three-years grant received has given her the possibility to investigate promising research issues at the Institute for Informatics, University of Munich, under the supervision of Prof. Dr. François Bry. Assisted by her supervisor, her research work has yielded the proposal found in this Ph.D. thesis. During the last years Ms. Paula-Lavinia Pătrânjan has published a couple of scientific articles in proceedings of conferences such as the Symposium of Applied Computing or the World Wide Web Conference.

Ms. Paula-Lavinia Pătrânjan is also actively involved in the Working Group I5 “Evolution and Reactivity” of the Network of Excellence “Reasoning on the Web with Rules and Semantics” (REWERSE), an EU FP6 project funded by the European Commission and by the Swiss Federal Office for Education and Science.

Ms. Paula-Lavinia Pătrânjan is very interested in dissemination and standardisation of research outcomes. Along this line, she is one of the co-organisers of the Workshop ’Reactivity on the Web’ to be held as an official satellite event at the International Conference on Extending Database Technology (EDBT 2006) in Munich, Germany.